

Eerke A. Boiten
John Derrick
Graeme Smith (Eds.)

LNCS 2999

Integrated Formal Methods

4th International Conference, IFM 2004
Canterbury, UK, April 2004
Proceedings



Springer

Lecture Notes in Computer Science

2999

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

Eerke A. Boiten John Derrick
Graeme Smith (Eds.)

Integrated Formal Methods

4th International Conference, IFM 2004
Canterbury, UK, April 4-7, 2004
Proceedings

eBook ISBN: 3-540-24756-4
Print ISBN: 3-540-21377-5

©2005 Springer Science + Business Media, Inc.

Print ©2004 Springer-Verlag
Berlin Heidelberg

All rights reserved

No part of this eBook may be reproduced or transmitted in any form or by any means, electronic, mechanical, recording, or otherwise, without written consent from the Publisher

Created in the United States of America

Visit Springer's eBookstore at:
and the Springer Global Website Online at:

<http://ebooks.springerlink.com>
<http://www.springeronline.com>

Preface

The fourth conference in the series of international meetings on Integrated Formal Methods, IFM, was held in Canterbury, UK, 4–7 April 2004. The conference was organized by the Computing Laboratory at the University of Kent, whose main campus is just outside the ancient town of Canterbury, part of the county of Kent.

Kent is situated in the southeast of England, and the university sits on a hill overlooking the city of Canterbury and its world-renowned cathedral. The University of Kent was granted its Royal Charter in 1965. Today there are almost 10,000 full-time and part-time students, with over 110 nationalities represented.

The IFM meetings have proven to be particularly successful. The first meeting was held in York in 1999, and subsequently we held events in Germany in 2000, and then Finland in 2002. The conferences are held every 18 months or so, and attract a wide range of participants from Europe, the Americas, Asia and Australia. The conference is now firmly part of the formal methods conference calendar. The conference has also evolved in terms of themes and subjects represented, and this year, in line with the subject as a whole, we saw more work on verification as some of the challenges in this subject are being met.

The work reported at IFM conferences can be seen as part of the attempt to manage complexity by combining paradigms of specification and design, so that the most appropriate design tools are used at different points in the life-cycle. In part this is about combining specification formalisms, and this happens, for example, when one combines state-based and event-based languages to produce integrated notations capable of covering a wider range of the design spectrum than would otherwise be feasible. However, the work of IFM goes beyond that, as we can see in this proceedings.

Indeed, increasingly specification is only the start of a process that includes verification as an explicit aim, and this work was heavily represented in this year's conference. This was also reflected in the talks by invited speakers who represented both academic and industry perspectives on the subject.

Tom Ball talked about his team's work on SLAM and the static driver verifier, and described the use of formal methods in Microsoft. Ursula Martin, of Queen Mary, University of London, talked about her work on design verification for control engineering, and Tom Melham of Oxford gave a talk entitled "Integrating Model Checking and Theorem Proving in a Reflective Functional Language." Tom Ball's talk was sponsored by FME (Formal Methods Europe), to whom we are particularly grateful. FME also held their Annual General Meeting on the Sunday prior to the main conference. We are also grateful to Jim Woodcock for agreeing to give a tutorial on the Unifying Theories of Programming, jointly with Ana Cavalcanti. FORTEST, a UK national network on Formal Methods and Testing, also joined us for the final day when members attended talks on testing and then held an informal workshop after the main conference.

The contributed talks were grouped into a number of sessions, this year covering:

- Automating program analysis
- State-/event-based verification
- Formalizing graphical notations
- Refinement
- Object orientation
- Hybrid and timed automata
- Integration frameworks
- Verifying interactive systems
- Testing and assertions

In total there were 65 submissions, of which we accepted 24 after the usual refereeing process. We are grateful to all those involved in the reviewing process and subsequent programme committee discussion. An important note of thanks must also be given to all those who helped locally.

We hope that these proceedings will serve as a useful source of reference for not only the attendees, but also the wider community. We look forward to further IFM meetings where we can continue the discussion on the best ways to engineer both hardware and software systems with the ultimate aim of increased reliability and robustness.

April 2004

Eerke Boiten
John Derrick
Graeme Smith

Program Committee

Didier Bert (France)
Eerke Boiten (Co-chair, UK)
Jonathan Bowen (UK)
Michael Butler (UK)
Paul Curzon (UK)
Jim Davies (UK)
John Derrick (Co-chair, UK)
Jin Song Dong (Singapore)
John Fitzgerald (UK)
Andrew Galloway (UK)
Chris George (Macau)
Wolfgang Grieskamp (US)
Henri Habrias (France)

Susumu Hayashi (Japan)
Maritta Heisel (Germany)
Michel Lemoine (France)
Shaoying Liu (Japan)
Dominique Méry (France)
Luigia Petre (Finland)
Judi Romijn (The Netherlands)
Thomas Santen (Germany)
Steve Schneider (UK)
Wolfram Schulte (US)
Kaisa Sere (Finland)
Jane Sinclair (UK)
Graeme Smith (Co-chair, Australia)

Bill Stoddart (UK)
 Kenji Taguchi (UK)
 W.J. (Hans) Toetenel
 (The Netherlands)

Heike Wehrheim (Germany)
 Kirsten Winter (Australia)
 Jim Woodcock (UK)

Sponsors

In addition to FME sponsorship of an invited talk, we are grateful to BCS FACS (the Formal Aspects of Computing Science Specialist Group of the British Computer Society, <http://www.bcs.org.uk/>) for sponsoring the best paper award.



External Referees

All submitted papers were reviewed by members of the program committee and a number of external referees, who produced extensive review reports and without whose work the conference would lose its quality status. To the best of our knowledge the list below is accurate. We apologize for any omissions or inaccuracies.

Bernhard K. Aichernig	Biniam Gebremichael	Pascal Poizat
Marcus Alanen	Michael Goldsmith	Mike Poppleton
Pascal André	Andy Gravell	Ivan Porres
Jim Armstrong	Stefan Hallerstede	Marie-Laure Potet
Mike Barnett	Ian Hayes	Viorel Preoteasa
Gerd Behrmann	Steffen Helke	Arend Rensink
Dag Björklund	Jon Jacky	Steve Riddle
Victor Bos	Nigel Jefferson	Michael Rusinowitch
Pontus Boström	Sara Kalvala	Gwen Salaün
Sylvain Boulmé	Maciej Koutny	Cristina Seculeanu
Robert Büsow	Yves Ledru	Dirk Seifert
Ana Cavalcanti	Hui Liang	Colin Snook
Orieta Celiku	Zhiming Liu	Mariëlle Stoelinga
Christine Choppy	Stephan Merz	Cedric Stoquer
Corina Cirstea	Pierre Michel	Carsten Sühl
Dang Van Hung	Arjan J. Mooij	Jun Sun
Henning Dierks	Mohammad Reza	Xinbei Tang
Roger Duke	Mousavi	Nikolai Tillmann
Steve Dunne	Catherine Oriat	Helen Treharne
Neil Evans	Stephen Paynter	Leonidas Tsiopoulos
Li Yuan Fang	Maria	Margus Veanes
Ansgar Fehnker	Pietkiewicz-Koutny	Sergiy Vilkomir
Leonardo Freitas	Juha Plosila	Marina Waldén

VIII Preface

Hai Wang
Virginie Wiels
Hirokazu Yatsu

Volker Zerbe
Frank Zeyda
Andrea Zisman

Steffen Zschaler

Table of Contents

Invited Talks

SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft	1
<i>Thomas Ball, Byron Cook, Vladimir Levin, Sriram K. Rajamani</i>	
Design Verification for Control Engineering.....	21
<i>Richard J. Boulton, Hanne Gottliebsen, Ruth Hardy, Tom Kelsey, Ursula Martin</i>	
Integrating Model Checking and Theorem Proving in a Reflective Functional Language	36
<i>Tom Melham</i>	

Tutorial

A Tutorial Introduction to Designs in Unifying Theories of Programming	40
<i>Jim Woodcock, Ana Cavalcanti</i>	

Contributed Papers

An Integration of Program Analysis and Automated Theorem Proving	67
<i>Bill J. Ellis, Andrew Ireland</i>	
Verifying Controlled Components	87
<i>Steve Schneider, Helen Treharne</i>	
Efficient CSP _Z Data Abstraction.....	108
<i>Adalberto Farias, Alexandre Mota, Augusto Sampaio</i>	
State/Event-Based Software Model Checking	128
<i>Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, Nishant Sinha</i>	
Formalising Behaviour Trees with CSP	148
<i>Kirsten Winter</i>	
Generating MSCs from an Integrated Formal Specification Language.....	168
<i>Jin Song Dong, Shengchao Qin, Jun Sun</i>	

UML to B: Formal Verification of Object-Oriented Models	187
<i>K. Lano, D. Clark, K. Androutsopoulos</i>	
Software Verification with Integrated Data Type Refinement for Integer Arithmetic	207
<i>Bernhard Beckert, Steffen Schlager</i>	
Constituent Elements of a Correctness-Preserving UML Design Approach	227
<i>Tiberiu Seceleanu, Juha Plosila</i>	
Relating Data Independent Trace Checks in CSP with UNITY Reachability under a Normality Assumption	247
<i>Xu Wang, A.W. Roscoe, R.S. Lazic</i>	
Linking CSP-OZ with UML and Java: A Case Study	267
<i>Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, Heike Wehrheim</i>	
Object-Oriented Modelling with High-Level Modular Petri Nets.....	287
<i>Cécils Bui Thanh, Hanna Klaudel</i>	
Specification and Verification of Synchronizing Concurrent Objects	307
<i>Gabriel Ciobanu, Dorel Lucanu</i>	
Understanding Object-Z Operations as Generalised Substitutions	328
<i>Steve Dunne</i>	
Embeddings of Hybrid Automata in Process Algebra	343
<i>Tim A.C. Willemse</i>	
An Optimal Approach to Hardware/Software Partitioning for Synchronous Model	363
<i>Pu Geguang, Dang Van Hung, He Jifeng, Wang Yi</i>	
A Many-Valued Logic with Imperative Semantics for Incremental Specification of Timed Models	382
<i>Ana Fernández Vilas, José J. Pazos Arias, Rebeca P. Díaz Redondo, Alberto Gil Solla, Jorge García Duque</i>	
Integrating Temporal Logics.....	402
<i>Yifeng Chen, Zhiming Liu</i>	
Integration of Specification Languages Using Viewpoints	421
<i>Marius C. Bujorianu</i>	
Integrating Formal Methods by Unifying Abstractions	441
<i>Raymond Boute</i>	

Formally Justifying User-Centred Design Rules:	
A Case Study on Post-completion Errors	461
<i>Paul Curzon, Ann Blandford</i>	
Using UML Sequence Diagrams as the Basis for a Formal Test	
Description Language	481
<i>Simon Pickin, Jean-Marc Jézéquel</i>	
Viewpoint-Based Testing of Concurrent Components	501
<i>Luke Wildman, Roger Duke, Paul Strooper</i>	
A Method for Compiling and Executing Expressive Assertions	521
<i>F.J. Galán Morillo, J.M. Cañete Valdeón</i>	
Author Index	541

This page intentionally left blank

SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft

Thomas Ball, Byron Cook, Vladimir Levin, and Sriram K. Rajamani

Microsoft Corporation

Abstract. The SLAM project originated in Microsoft Research in early 2000. Its goal was to automatically check that a C program correctly uses the interface to an external library. The project used and extended ideas from symbolic model checking, program analysis and theorem proving in novel ways to address this problem. The SLAM analysis engine forms the core of a new tool called Static Driver Verifier (SDV) that systematically analyzes the source code of Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows operating system kernel.

We believe that the history of the SLAM project and SDV is an informative tale of the technology transfer of formal methods and software tools. We discuss the context in which the SLAM project took place, the first two years of research on the SLAM project, the creation of the SDV tool and its transfer to the Windows development organization. In doing so, we call out many of the basic ingredients we believe to be essential to technology transfer: the choice of a critical problem domain; standing on the shoulders of those who have come before; the establishment of relationships with “champions” in product groups; leveraging diversity in research and development experience and careful planning and honest assessment of progress towards goals.

1 Introduction

In the early days of computer science, the ultimate goal of formal methods and program verification was to provide technology that could rigorously prove programs fully correct. While this goal remains largely unrealized, many researchers now focus on the less ambitious but still important goal of stating *partial* specifications of program behavior and providing methodologies and tools to check their correctness. The growing interest in this topic is due to the technological successes and convergence of four distinct research areas—type checking, model checking, program analysis, and automated deduction—on the problems of software quality. Ideas about specification of properties, abstraction of programs, and algorithmic analyses from these four areas are coming together in new ways to address the common problem of software quality.

The SLAM¹ project is just one of many exploring this idea. In early 2000 we set out to build a software tool that could automatically check that a C program correctly uses the interface to an external library. The outcome of this project is the SLAM analysis engine, which forms the core of a soon-to-be-released tool called Static Driver Verifier (SDV). SDV systematically analyzes the source code of Windows device drivers against a set of rules that define what it means for a device driver to properly interact with the Windows kernel, the heart of the Windows operating system (referred to as “Windows” from now on). In effect, SDV tests all possible execution paths through the C code.

To date, we have used SDV internally to find defects in Microsoft-developed device drivers, as well as in the sample device drivers that Microsoft provides in the Windows Driver Development Kit (DDK). However, the most important aspect of Windows’s stability is the quality of the device drivers written outside of Microsoft, called *third-party drivers*. For this reason we are now preparing SDV for release as part of the DDK.

We have written many technical research papers about SLAM but we have never before written a history of the non-technical aspects of the project. Our goal is to discuss the process of technology transfer from research to development groups and to highlight the reasons we believe that we have been successful to date, some of which are:

- *Choice of Problem*: We chose a critical, but not insurmountable, problem domain to work on (device drivers). We had access to the Windows source code and the source code of the device drivers. We also had extensive access to the foremost experts on device drivers and Windows.
- *Standing on Shoulders*: SLAM builds on decades of research in formal methods and programming languages. We are fortunate to have had many people contribute to SLAM and SDV, both in Microsoft Research, the Windows division, as well as from outside Microsoft.
- *Research Environment*: Microsoft’s industrial research environment and general “hands-on/can-do” culture allowed us great freedom in which to attempt a risky solution to a big problem, and provided support when we needed it the most.
- *Software Engineering*: We developed SLAM in an “open” architectural style using very simple conceptual interfaces for each of its core components. This allowed us to experiment quickly with various tools and settle on a set of algorithms that we felt best solved the problem. This architecture also allows us to reconfigure the various components easily in response to new problems.
- *The Right Tools for the Job*: We developed SLAM using INRIA’s OCaml functional programming language. The expressiveness of this language and robustness of its implementation provided a great productivity boost.
- *Good Luck*: We experienced good luck at many points over the past four years and fortunately were able to take advantage of it.

¹ SLAM originally was an acronym but we found it too cumbersome to explain. We now prefer to think of “slamming” the bugs in a program.

While some of these factors may be unique to our situation, many are the basic ingredients of successful research, development, and technology transfer. We believe that the history of our project makes an interesting case study in the technology transfer of formal methods and software tools in industry.

We tell the story in four parts. Section 2 discusses the context in which the SLAM and SDV projects took place. In particular, this section provides background on Windows device drivers and Microsoft Research. Section 3 discusses the first two years of the SLAM project, when the bulk of the research took place. Section 4 discusses the development of the Static Driver Verifier tool and its transfer to the Windows development organization. Section 5 concludes with an analysis of the lessons we learned from our four year experience and a look at the future.

2 Prologue

We will now provide some pre-SLAM history so that the reader will better understand the context in which our project originated.

2.1 Windows Device Drivers

Windows hides from its users many details of the myriad hardware components that make up a personal computer (PC). PCs are assembled by companies who have purchased many of the PC's basic components from other companies. The power of Windows is that application programmers are still able to write programs that work using the interface provided by Windows with little to no concern for the underlying hardware that their software eventually will execute on.

Examples of devices include keyboards, mice, printers, graphics and audio cards, network interface cards, cameras, and a number of storage devices, such as CD and DVD drives. Device drivers are the software that link the component devices that constitute a PC, as well as its peripheral devices, to Windows. The number of devices and device drivers for Windows is enormous, and grows every day. While only about 500 device drivers ship on a Windows CD, data collected through Microsoft's Online Crash Analysis (OCA) tool shows orders of magnitude more device drivers deployed in the field.

Most device drivers run within the Windows kernel, where they can run most efficiently. Because they execute in the kernel, poorly written device drivers can cause the Windows kernel (and thus the entire operating system) to crash or hang. Of course, such device driver failures are perceived by the end-user as a failure of Windows, not the device driver. Driver quality is a key factor in the Windows user experience and has been a major source of concern within the company for many years.

The most fundamental interface that device drivers use to communicate with the Windows kernel is called the Windows Driver Model (WDM). As of today,

this interface includes over 800 functions providing access to various kernel facilities: memory allocation, asynchronous I/O, threads, events, locking and synchronization primitives, queues, deferred procedure calls, interrupt service routines, etc. Various classes of drivers (network drivers, for example) have their own driver models, which provide device-specific interfaces on top of the WDM to hide its complexity.

Microsoft provides the Driver Development Kit (DDK) to aid third-parties in writing device drivers. The DDK contains the Microsoft compiler for the C and C++ languages, supporting tools, documentation of the WDM and other driver models, and the full source code of many drivers that ship on the Windows CD. The DDK also contains a number of software tools specifically oriented towards testing and analyzing device drivers. One is a tool called Driver Verifier, which finds driver bugs while the drivers execute in real-time in Windows. In addition to the DDK, Microsoft has a driver certification program whose goal is to ensure that drivers digitally signed by Microsoft meet a certain quality bar. Finally, Microsoft uses the OCA feature of Windows to determine which device drivers are responsible for crashes in the field. This data is made available to Microsoft's partners to ensure that error-prone drivers are fixed as quickly as possible. Despite all these measures, drivers are a continuing source of errors. Developing drivers using a complex legacy interface such as WDM is just plain hard. (This is not just true of Windows—Engler found the error rate in Linux device drivers was much higher than for the rest of the Linux kernel [CYC⁺01]).

Device drivers are a great problem domain for automated analysis because they are relatively small in size (usually less than 100,000 lines of C code), and because most of the WDM usage rules are control-dominated and have little dependence on data. On the other hand, drivers use all the features of the C language and run in a very complex environment (the Windows kernel), which makes for a challenging analysis problem.

One of the most difficult aspects of doing work in formal methods is the issue of where specifications come from, and the cost of writing and maintaining them. A welcome aspect of the WDM interface, from this perspective, is that the cost of writing the specifications can be amortized by checking the same specifications over many WDM drivers. Interfaces that are widely used (such as the WDM) provide good candidates for applying formal methods, since specifications can be done at the level of the interface and all clients that use the interface can be analyzed automatically for consistent usage of the interface with respect to the specifications.

2.2 Microsoft Research

Over the past decade, Microsoft Research (MSR) has grown to become one of the major industrial research organizations in basic computer science, with over 600 researchers in five labs worldwide.

It is worthwhile to note the major differences between industrial research, as found in Microsoft, and research at academic institutions. First, there is no tenure in MSR, as in academia. Performance reviews take place every year, as

done in corporate America. Second, performance is measured not only by contributions to basic science (one measure of which is peer-reviewed publications) but also by contributions to Microsoft. Balancing long-term basic research with more directed work for the company is one of the most challenging but also the most rewarding aspects of industrial research. Third, working with other researchers within MSR (as well as outside) is encouraged and rewarded. Fourth, there are no graduate students. Instead, during three brief summer months each year, we are fortunate to attract high quality graduate students for internships. One final thing is worth noting: MSR generally puts high value on seeing ideas take form in software, as this is the major mechanism for demonstrating value and enabling technology transfer within Microsoft. To say this in a different way: developers are not the only Microsoft employees who program computers; researchers also spend a good deal of time creating software to test their ideas. As we discovered in SLAM, new research insights often come from trying to take an idea from theory to practice through programming.

The Programmer Productivity Research Center (PPRC) is a research and development center in MSR whose charter is “to radically improve the effectiveness of software development and the quality of Microsoft software”. Founded in March of 1999, PPRC’s initial focus was on performance tools but quickly grew to encompass reliability tools with the acquisition of Intrinsa and its PREFIX defect detection tool [BPS00]. The PREFIX technology has been deployed in many of Microsoft’s product groups. More than twelve percent of the bugs fixed before Windows 2003 server shipped were found with the PREFIX and PREFast tools, which are run regularly over the entire Windows source base. PPRC has developed an effective infrastructure and pipeline for developing new software tools and deploying them throughout the company.

3 SLAM (2000–2001)

So, the stage is set to tell the story of SLAM. Device drivers were (and still are) a key problem of concern to the company. PPRC, which supports basic research in programming languages, formal methods and software engineering, was seeking to improve development practices in Microsoft through software tools. In this section, we describe the first two years of the SLAM project.

3.1 Software Productivity Tools

SLAM was one of the initial projects of the Software Productivity Tools (SPT) group within PPRC, founded by Jim Larus. The members of this group were Tom Ball, Manuvir Das, Rob DeLine, Manuel Fähndrich, Jim Larus, Jakob Rehof and Sriram Rajamani. The SPT group spent its first months brainstorming new project ideas and discussing software engineering problems. The problem of device drivers was one of the topics that we often discussed.

Three projects came out of these discussions: SLAM, Vault [DF01], and ESP [DLS02]. Each of these projects had a similar goal: to rigorously check that

a program obeys “interface usage rules”. The basic differences in the projects were in the way the rules were specified and in the analysis technology used. Vault was a new programming language with an extended type system in which the rules were specified using pre-/post-conditions attached to types. ESP and SLAM shared a similar specification language but took different approaches to addressing the efficiency/precision tradeoffs inherent in program analysis. (For a more detailed comparison of these three projects, see [LBD⁺04].)

Having several projects working in friendly competition on a common problem made each project stronger. We benefited greatly from many technical discussions with SPT members. All three projects are still active today: Manuvir now leads a group based on the ESP project to extend the scope and scale of static analysis tools; Rob and Manuel retargeted the Vault technology to MSIL (Microsoft’s Intermediate Language, a byte-code like language for Microsoft’s new virtual machine, the Common Language Runtime) and extended its capabilities. This analyzer is called Fugue [DF04] and is a plug-in to the Visual Studio programming environment and will be available soon as part of the freely-available FxCop tool.

3.2 A Productive Peer Partnership

SLAM was conceived as the result of conversations between Tom and Sriram on how symbolic execution, model checking and program analysis could be combined to solve the interface usage problem for C programs (and drivers in particular). Tom’s background was in programming languages and program analysis, while Sriram’s background was in hardware verification and model checking. Both had previous experience in industry. Tom worked six years as a researcher in Bell Labs (at AT&T and then Lucent Technologies) after his Ph.D. and Sriram worked over five years at Syntek and Xilinx before his Ph.D. Two months of initial discussions and brainstorming at the end of 1999 led to a technical report published in January of 2000 [BR00b] that contained the basic ideas, theory and algorithms that provided the initial foundation for the SLAM project.

Our basic idea was that checking a simple rule against a complex C program (such as a device driver) should be possible by simplifying the program to make analysis tractable. That is, we should be able to find an *abstraction* of the original C program that would have all of the behaviors of the original program (plus additional ones that did not matter when checking the rule of interest).

The basic question we then had to answer was “What form should an abstraction of a C program take?”. We proposed the idea of a Boolean program, which would have the same control flow structure as the original C program but only permit the declaration of Boolean variables. These Boolean variables would track important predicates over the original program’s state (such as $x < 5$).

We found Boolean programs interesting for a number of reasons. First, because the amount of storage a Boolean program can access at any point is finite, questions of reachability and termination (which are undecidable in general) are decidable for Boolean programs. Second, as Boolean programs contain the control-flow constructs of C, they form a natural target for investigating model

checking of software. Boolean programs can be thought of as an abstract representation of C programs in which the original variables are replaced by Boolean variables that represent relational observations (predicates) between the original variables. As a result, Boolean programs are useful for reasoning about properties of the original program that are expressible through such observations.

Once we fixed Boolean programs as our form of abstraction, this led us naturally to an automated process for abstraction, checking and refinement of Boolean programs in the spirit of Kurshan [Kur94]:

- **Abstract.** Given a C program P and set of predicates E , the goal of this step is to efficiently construct a precise Boolean program abstraction of P with respect to E . Our contribution was to extend the predicate abstraction algorithm of Graf and Saïdi [GS97] to work for programs written in common programming languages (such as C).
- **Check.** Given a Boolean program with an error state, the goal of this step is to check whether or not the error state is reachable. Our contribution was to solve this problem by using a data structure called Binary Decision Diagrams [Bry86,BCM⁺92] from the model checking community in the context of traditional interprocedural dataflow analysis [SP81,RHS95].
- **Refine.** If the Boolean program contains an error path and this path is a feasible execution path in the original C, then the process has found a potential error. If this path is not feasible in the C program then we wish to refine the Boolean program so as to eliminate this false error path. Our contribution was to show how to use symbolic execution and a theorem prover [DNS03] to find a set of predicates that, when injected into the Boolean program on the next iteration of the SLAM process, would eliminate the false error path.

In the initial technical report, we formalized the SLAM process and proved its soundness for a language with integer variables, procedures and procedure calls but without pointers. Through this report we had laid out a plan and a basic architecture that was to remain stable and provide a reference point as we progressed. Additionally, having this report early in the life of the project helped us greatly in recruiting interns. The three interns who started on the SLAM project in the summer of 2000 had already digested and picked apart the technical report before they arrived.

After we had written the technical report we started implementing the **Check** step in the BEBOP model checker [BR00a,BR01a]. Although only one of the three steps in SLAM was implemented, it greatly helped us to explore the SLAM process as we could simulate the other two steps by hand (for small examples). Furthermore, without the **Check** step, we could not test the **Abstract** step, which we planned to implement in the summer.

During the implementation of BEBOP, we often worked side-by-side as we developed code. We worked to share our knowledge about our respective fields, program languages/analysis (Tom) and model checking (Sriram). Working in this fashion, we had an initial implementation of BEBOP working in about two months.

With only BEBOP working, we manually extracted Boolean program models from several drivers and experimented with the entire approach. Then, over the summer of 2000, we built the first version of the **Abstract** step with the help of our interns Rupak Majumdar and Todd Millstein. After this was done, we experimented with more examples where we manually supplied predicates, but automatically ran the **Abstract** and **Check** steps. Finally, in the fall of 2000, we built the first version of the **Refine** step. Since this tool discovers predicates we named it NEWTON [BR02a]. We also developed a language called SLIC to express interface usage rules in a C-like syntax, and integrated it with the rest of the tools [BR01b].

3.3 Standing on Shoulders

As we have mentioned before, the ideas that came out of the SLAM project built on and/or extended previous results in the areas of program analysis, model checking and theorem proving. A critical part to SLAM’s success was not only to build on a solid research foundation but also to build on existing technology and tools, and to enlist other people to help us build and refine SLAM.

The parts of SLAM that analyze C code were built on top of existing infrastructure developed in MSR that exports an abstract syntax tree interface from the Microsoft C/C++ compiler and that performs alias analysis of C code [Das00]. The BEBOP model checker uses a BDD library called CUDD developed at The University of Colorado [Som98]. (This library also has been incorporated in various checking tools used within Intel and other companies that develop and apply verification technology.) We also relied heavily on the Simplify theorem prover from the DEC/Compaq/HP Systems Research Center [DNS03]. Finally, the SLAM code base (except for the BEBOP model checker) was written in the functional programming language Objective Caml (O’Caml) from INRIA [CMP]. BEBOP was written in C++.

In our first summer we were fortunate to have three interns work with us on the SLAM project: Sagar Chaki from Carnegie Mellon University (CMU), Rupak Majumdar from the University of California (UC) at Berkeley and Todd Millstein from the University of Washington. Rupak and Todd worked on the first version of the predicate abstraction tool for C programs [BMMR01], while Sagar worked with us on how to reason about concurrent systems [BCR01]. After returning to Berkeley, Rupak and colleagues there started the BLAST project, which took a “lazy” approach to implementing the process we had defined in SLAM [HJMS02]. Todd continued to work with us after the summer to finish the details of performing predicate abstraction in the presence of procedures and pointers [BMR01]. Back at CMU, Sagar started the MAGIC project [CCG⁺03], which extended the ideas in SLAM to the domain of concurrent systems.

During these first two years, we also had the pleasure of hosting other visitors from academia. Andreas Podelski, from the Max Plank Institute, spent his sabbatical at MSR and helped us understand the SLAM process in terms of abstract interpretation [CC77]. Andreas’ work greatly aided us in understanding the theoretical capabilities and limitations of the SLAM process [BPR01,

BPR02]. Stefan Schwoon, a Ph.D. candidate from the Technical University of München, visited us in the fall of 2001. Stefan had been working on a model checking tool [ES01]—called MOPED—that was similar to BEBOP. We had sent him information about Boolean programs, which allowed him to target MOPED to our format. In a few weeks of work with us, he had a version of SLAM that worked with MOPED instead of BEBOP. As a result, we could directly compare the performance of the two model checkers. This led to a fruitful exchange of ideas about how to improve both tools.

Later on, Rustan Leino joined the SPT group and wrote a new Boolean program checker (called “Dizzy”) that was based on translating Boolean programs to SAT [Lei03]. This gave us two independent ways to analyze Boolean programs and uncovered even more bugs in BEBOP.

Finally, as we mentioned before, the PREFIX and PREFast tools blazed the trail for static analysis at Microsoft. These two tools have substantially increased the awareness within the company of the benefits and limitations of program analysis. The success of these tools has made it much easier for us to make a case for the next generation of software tools, such as SDV.

3.4 Champions

A key part of technology transfer between research and development organizations is to have “champions” on each side of the fence. Our initial champions in the Windows organization were Adrian Oney, Peter Wieland and Bob Rinne.

Adrian is the developer of the Driver Verifier testing tool built into the Windows operating system (Windows 2000 and on). Adrian spent many hours with us explaining the intricacies of device drivers. He also saw the potential for Static Driver Verifier to complement the abilities of Driver Verifier, rather than viewing it as a competing tool, and communicated this potential to his colleagues and management. Peter Wieland is an expert in storage drivers and also advised us on the complexities of the driver model. If we found what we thought might be a bug using SLAM, we would send email to Adrian and Peter. They would either confirm the bug or explain why this was a false error. The latter cases helped us to refine the accuracy of our rules. Additionally, Neill Clift from the Windows Kernel team had written a document called “Common Driver Reliability Problems” from which we got many ideas for rules to check.

Having champions like these at the technical level is necessary but not sufficient. One also needs champions at the management level with budgetary power (that is, the ability to hire people) and the “big picture” view. Bob Rinne was our champion at the management level. Bob is a manager of the teams responsible for developing many of device drivers and driver tools that Microsoft ships. As we will see later, Bob’s support was especially important for SLAM and SDV to be transferred to Windows.

3.5 The First Bug ... and Counting

In initial conversations, we asked Bob Rinne to provide us with a real bug in a real driver that we could try to discover with the SLAM engine. This would be the first test of our ideas and technology. He presented us with a bug in the floppy disk driver from the DDK that dealt with the processing of IRPs (I/O Request Packets). In Windows, requests to drivers are sent via IRPs. There are several rules that a driver must follow with regards to the management of IRPs. For instance, a driver must mark an IRP as pending (by calling `IoMarkIrpPending`) if it returns `STATUS_PENDING` as the result of calling the driver with that IRP. The floppy disk driver had one path through the code where the correlation between returning `STATUS_PENDING` and calling `IoMarkIrpPending` was missed. On March 9, 2001, just one year after we started implementing SLAM, the tool found this bug.

In the summer of 2001, we were again fortunate to have excellent interns working on the SLAM project: Satyaki Das from Stanford, Sagar Chaki (again), Robby from Kansas State University and Westley Weimer from UC Berkeley. Satyaki and Westley worked on increasing the performance of the SLAM process [ABD⁺02,BCDR04] and the number of device drivers to which we could successfully apply SLAM. Robby worked with Sagar on extending SLAM to reason more accurately about programs which manipulate heap data structures. Towards the end of the summer Westley and Satyaki found two previously unknown bugs in DDK sample drivers using SLAM.

Manuel Fähndrich developed a diagram of the various legal states and transitions an IRP can go through by piecing together various bits of documentation, and by reading parts of the kernel source code. Using this state diagram, we encoded a set of rules for checking IRP state management. With these rules we found five more previously unknown bugs in IRP management in various drivers.

3.6 Summary

In the first two years of the SLAM project we had defined a new direction for software analysis based on combining and extending results from the fields of model checking, program analysis and theorem proving, published a good number of papers (see references for a full list), created a prototype tool that found some non-trivial bugs in device drivers, and had attracted attention from the academic research community. The first two years culminated in an invited talk which we were asked to present at the Symposium on the Principles of Programming Languages in January of 2002 [BR02b].

However, as we will see, the hardest part of our job was still ahead of us. As Thomas Alva Edison noted, success is due in small part to “inspiration” and in large part to “perspiration”. We had not yet begun to sweat.

4 Static Driver Verifier (2002-2003)

From an academic research perspective, SLAM was a successful project. But, in practice, SLAM could only be applied productively by a few experts. There

was a tremendous amount of work left to do so that SLAM could be applied automatically to large numbers of drivers. In addition to improving the basic SLAM engine, we needed to surround this engine with the framework that would make it easy to run on device drivers. The product that solved all of these problems was to be called “Static Driver Verifier” (SDV). Our vision was to make SDV a fully automatic tool. It had to contain, in addition to the SLAM engine, the following components:

- A large number of rules for the Windows Driver Model (and in future releases, other driver models as well)—we had written only a handful of rules;
- A model of the Windows kernel and other drivers, called the environment model—we had written a rough model of the environment model in C, but it needed to be refined;
- Scripts to build a driver and configure SDV with driver specific information;
- A graphical user interface (GUI) to summarize the results of running SDV and to show error traces in the source code of the driver.

SDV was not going to happen without some additional help.

Having produced promising initial results, we went to Amitabh Srivastava, director of the PPRC, and asked for his assistance. He committed to hiring a person for the short term to help us take SLAM to the next stage of life. Fortunately, we had already met just the right person for the task: Jakob Lichtenberg from the IT University of Copenhagen. We met Lichtenberg in Italy at the TACAS conference in 2001 where we presented work with our summer interns from 2000. After attending our talk, Jakob had spent the entire night re-coding one of our algorithms in a model checking framework he had developed. We were impressed. Lichtenberg joined the SLAM team in early February of 2002 and the next stage of the roller-coaster ride began. Jakob was originally hired for six months. In the end, he stayed 18 months.

4.1 TechFest and Bill Gates Review

The first task Lichtenberg helped us with was preparing a demonstration for an internal Microsoft event in late February of 2002 called TechFest. TechFest is an annual event put on by MSR to show what it has accomplished in the past year and to find new opportunities for technology transfer. TechFest has been an incredibly popular event. In 2001, when TechFest started, it had 3,700 attendees. In its second year, attendance jumped to 5,200. In 2003, MSR’s TechFest was attended by over 7,000 Microsoft employees.

The centerpiece of TechFest is a demo floor consisting of well over 100 booths. In our booth, we showed off the results of running SLAM on drivers from the Driver Development Kit of Windows XP. Many driver developers dropped by for a demo. In some cases, the author of a driver we had found a bug in was present to confirm that we had found a real bug. Additionally, two other important people attended the demo: Jim Allchin (head of the Windows platform division) and Bill Gates.

Two weeks after TechFest (in early March 2002), we made a presentation on SLAM as part of a regular review of research by Bill Gates. At this point, managers all the way up the management chain in both MSR and Windows (with the least-common ancestor being Gates) were aware of SLAM. The rapidity with which key people in the company became aware of SLAM and started referring to it was quite overwhelming.

4.2 The Driver Quality Team

Around this time, a new team in Bob Rinne’s organization formed to focus on issues of driver quality. Bob told us that he might be able to hire some people into this group, called the *Driver Quality Team* (DQT), to help make a product out of SDV. In the first four months of 2002, we had received a number of resumes targeted at the SLAM project. We told Bob of two promising applicants: Byron Cook, from the Oregon Graduate Institute (OGI) and Prover Technology, and Vladimir Levin, from Bell Labs. Byron was in the process of finishing his Ph.D. in Computer Science and had been working on tools for the formal verification of hardware and aircraft systems at Prover for several years. Vladimir had a Ph.D. in Computer Science and had been working on a formal verification tool at Bell Labs for six years.

By the beginning of July, both Byron and Vladimir were interviewed and hired. They would join Microsoft in August and September of 2002, respectively, as members of DQT. The importance of the Windows kernel development organization hiring two Ph.D.s with formal verification backgrounds and experience cannot be overstated. It was another major milestone in the technology transfer of SLAM. Technology transfer often requires transfer of expertise in addition to technology. Byron and Vladimir were to form the bridge between research and development that would enable SLAM to be more successful.

Nar Ganapathy was appointed as the manager of DQT. Nar is the developer and maintainer of the I/O subsystem of the Windows kernel — the piece of the kernel that drivers interact with most. This meant that half of the SDV team would now be reporting directly to the absolute expert on the behavior of the I/O subsystem.

4.3 SDV 1.0

Our first internal release of SDV (1.0) was slated for the end of the summer. This became the major focus of our efforts during the late spring and summer of 2002. While in previous years, summer interns had worked on parts of the SLAM engine, we felt that the analysis engine was stable enough that we should invest energy in problems of usability. Mayur Naik from Purdue University joined as a summer intern and worked on how to localize the cause of an error in an error trace produced by SLAM [BNR03].

On September 03, 2002, we made the release of SDV 1.00 on an internal website. It had the following components: the SLAM engine, a number of interface

usage rules, a model of the kernel used during analysis, a GUI and scripts to build the drivers.

4.4 Fall 2002: Descent into Chaos (SDV 1.1)

In the autumn of 2002, the SDV project became a joint project between MSR and Windows with the arrival of Byron and Vladimir, who had been given offices in both MSR and Windows. While we had already published many papers about SLAM, there was a large gap between the theory we published and the implementation we built. The implementation was still a prototype and was fragile. It only had been run on about twenty drivers. We had a small set of rules. Dependence on a old version of the Microsoft compiler and fundamental performance issues prevented us from running on more drivers.

When Byron and Vladimir began working with the system they quickly exposed a number of significant problems that required more research effort to solve. Byron found that certain kinds of rules made SLAM choke. Byron and Vladimir also found several of SLAM's modules to be incomplete. At the same time, a program manager named Johan Marien from Windows was assigned to our project part-time. His expectation was that we were done with the research phase of the project and ready to be subjected to the standard Windows development process. We were not ready. Additionally, we were far too optimistic about the timeframe in which we could address the various research and engineering issues needed to make the SLAM engine reliable. We were depending on a number of external components: O'Caml, the CUDD BDD package, the automatic theorem prover Simplify. Legal and administrative teams from the Windows organization struggled to figure out the implications of these external dependencies.

We learned several lessons in this transitional period. First, code reviews, code refactoring and cleanup activities provide a good way to educate others about a new code base while improving its readability and maintainability. We undertook an intensive series of meetings over a month and a half to review the SLAM code, identify problems and perform cleanup and refactoring to make the code easier to understand and modify. Both Byron and Vladimir rewrote several modules that were not well understood or buggy. Eventually, ownership of large sections of code was transferred from Tom and Sriram to Byron and Vladimir. Second, weekly group status meetings were essential to keeping us on track and aware of pressing issues. Third, it is important to correctly identify a point in a project where enough research has been done to take the prototype to product. We had not yet reached that point.

4.5 Winter 2002/Spring 2003: SDV Reborn (SDV 1.2)

The biggest problem in the autumn of 2002 was that a most basic element was missing from our project, as brought to our attention by Nar Ganapathy: we were lacking a clear statement of how progress and success on the SDV project would be measured. Nar helped us form a "criteria document" that we could use

to decide if SDV was ready for widespread use. The document listed the type of drivers that SDV needed to run on, specific drivers on which SDV needed to run successfully, some restrictions on driver code (initial releases of SDV were not expected to support C++), performance expectations from SDV (how much memory it should take, how much time it should take per driver and per rule), and the allowable ratio of false errors the tool could produce (one false error per four error reports).

Another problem was that we now had a project with four developers and no testers. We had a set of over 200 small regression tests for the SLAM engine itself, but we needed more tests, particularly with complete device drivers. We desperately needed better regression testing. Tom and Vladimir devoted several weeks to develop regression test scripts to address this issue. Meanwhile Byron spent several weeks convincing the Windows division to devote some testing resources to SDV. As a result of his pressure, Abdullah Ustuner joined the SDV team as a tester in February 2003.

One of the technical problems that we encountered is called NDF, an internal error message given by SLAM that stands for “no difference found”. This happens when SLAM tries to eliminate a false error path but fails to do so. In this case, SLAM halts without having found a true error or a proof of correctness. A root cause of many of these NDFs was SLAM’s lack of precision in handling pointer aliasing. This led us to invent novel ways to handle pointer aliasing during counter-example-driven refinement, which we implemented. SLAM also needed to be started with a more precise model of the kernel and possible aliases inside kernel data structures, so we rewrote the kernel models and harnesses to initialize key data structures. As a result of these solutions, the number of NDFs when we shipped SDV 1.2 went down dramatically. Some still remained, but the above solutions converted the NDF problem from a show-stopper to a minor inconvenience.

With regression testing in place, a clear criterion from Nar’s document on what we need to do to ship SDV 1.2, and reduction of the NDF problem, we slowly recovered from the chaos that we experienced in the winter months. SDV 1.2 was released on March 31st, 2003, and it was the toughest release we all endured. It involved two organizations, two different cultures, lots of people, and very hard technical problems. We worked days, nights and weekends to make this release happen.

4.6 Taking Stock in the Project: Spring 2003

Our group had been hearing conflicting messages about what our strategy should be. For example, should we make SDV work well on third party drivers and release SDV as soon as possible, or should we first apply it widely on our own internally developed drivers and find the most bugs possible? Some said we should take the first option; others said the latter option was more critical. Our group also needed more resources. For example, we needed a full-time program manager who could manage the legal process and the many administrative complications involved in transferring technology between organizations. We desper-

ately needed another tester. Additionally, we needed to get a position added in the Windows division to take over from Jakob, whose stay at Microsoft was to end soon.

Worst of all, there was a question as to whether SDV had been successful or not. From our perspective, the project had been a success based on its reception by the formal verification research community and MSR management. Some people within the Windows division agreed. Other members of the Windows division did not. The vast majority of people in the Windows division were not sure and wanted someone else to tell them how they should feel.

Byron decided that it was time to present our case to the upper management of the Windows division and worked with Nar to schedule a project review with Windows vice-president Rob Short. We would show our hand and simply ask the Windows division for the go-ahead to turn SDV into a product. More importantly, a positive review from Rob would help address any lingering doubts about SDV's value within his organization.

We presented our case to Rob, Bob Rinne and about ten other invited guests on April 28th 2003. We presented statistics on the number of bugs found with SDV and the group's goals for the next release: we planned on making the next release available at the upcoming Windows Driver Development Conference (DDC), where third-party driver writers would apply SDV to their own drivers. We made the case for hiring three more people, (a program manager, another tester and developer to take over from Jakob) and buying more machines to parallelize runs of SDV. In short order, Rob gave the "thumbs-up" to all our plans. It was time to start shopping for people and machines.

4.7 Summer/Fall 2003: The Driver Developer Conference (SDV 1.3)

Ideally we would have quickly hired our new team-members, bought our machines and then began working on the next release. However, it takes time to find the right people, as we found out. At the end of May, John Henry joined the SDV group as our second tester. Bohus Ondrusek would eventually join the SDV team as our program manager in September. Con McGarvey later joined as a developer in late September. Jakob Lichtenberg left to return to Denmark at about the same time. By the time we had our SDV 1.3 development team put together, the Driver Developer Conference was only a month away.

Meanwhile, we had been busy working on SLAM. When it became clear that we would not know if and when our new team-members would join, we decided to address the following critical issues for the DDC event:

- More expressiveness in the SLIC rule language.
- More rules. We added more than 60 new rules that were included in the DDC distribution of SDV.
- Better modeling of the Windows kernel. While not hoping to complete our model of the kernel by the DDC, we needed to experiment with new ways to generate models. A summer intern from the University of Texas at Austin

named Fei Xie spent the summer trying a new approach in which SLAM’s analysis could be used to *train* with the real Windows code and find a model that could be saved and then reused [BLX04]. Abdullah wrote a tool that converted models created by PREFIX for use by SLAM.

- Better integration with the “driver build” environment used by driver writers. This included supporting libraries and the new C compiler features used by many drivers.
- Removal of our dependency on the Simplify theorem prover. SLAM uses a first-order logic theorem prover during the **Abstract** and **Refine** steps described in Section 3.2. Up until this time we had used Simplify. But the license did not allow us to release SLAM based on this prover. Again, we relied on the help of others. Shuvendu Lahiri, a graduate student from CMU with a strong background in theorem proving, joined us for the summer to help create a new theorem prover called “Zapato”. We also used a SAT solver created by Lintao Zhang of MSR Silicon Valley. By the fall of 2003, we had replaced Simplify with Zapato in the SLAM engine, with identical performance and regression results. [BCLZ04]

In the end, the release of SDV 1.3 went smoothly. We released SDV 1.3 on November 5th, a week before the DDC. The DDC event was a great success. Byron gave two presentations on SDV to packed rooms. John ran two labs in which attendees could use SDV on their own drivers using powerful AMD64-based machines. Almost every attendee found at least one bug in their code. The feedback from attendees was overwhelmingly positive. In their surveys, the users pleaded with us to make a public release of SDV as soon as possible.

The interest in SDV from third-party developers caused even more excitement about SDV within Microsoft. Some of the attendees of the DDC were Microsoft employees who had never heard of SDV. After the DDC we spent several weeks working with new users within Microsoft. The feedback from the DDC attendees also helped us renew our focus on releasing SDV. Many nice features have not yet been implemented. On some drivers the performance could be made much better. But, generally speaking, the attendees convinced us (while the research in this class of tools is not yet done) that we have done enough research in order to make our first public release.

4.8 Summary

As of the beginning of 2004, the SDV project has fully transferred from Microsoft Research to Windows. There are now six people working full-time on SDV in Windows: Abdullah, Bohus, Byron, Con, John and Vladimir. Sriram and Tom’s involvement in the project has been reduced to “consultancy”; they are no longer heavily involved in the planning or development of the SLAM/SDV technology but are continuing research that may eventually further impact SDV .

5 Epilogue: Lessons Learned and the Future

We have learned a number of lessons from the SLAM/SDV experience:

- *Focus on Problems not Technology.* It is easier to convince a product group to adopt a new solution to a pressing problem that they already have. It is very hard to convince a product group to adopt new technology if the link to the problem that it solves is unclear. Concretely, we do not believe that trying to transfer the SLAM engine as an analysis vehicle could ever work. However, SDV as a solution to the driver reliability problem is an easier concept to sell to a product group (We thank Jim Larus for repeatedly emphasizing the important difference between problem and solution spaces).
- *Exploit Synergies.* It was the initial conversations between Tom and Sriram that created the spark that became the SLAM project. We think it is a great idea for people to cross the boundaries of their traditional research communities to collaborate with people from other communities and to seek diversity in people and technologies when trying to solve a problem. We believe that progress in research can be accelerated by following this recipe.
- *Plan Carefully.* As mentioned before, research is a mix of a small amount of inspiration and a large amount of perspiration. To get maximum leverage in any research project, one has to plan in order to be successful. In the SLAM project, we have spent long hours planning intern projects and communicating with interns long before they even showed up at MSR. We think that it is crucial not to underestimate the value of such ground work. Usually, we have had clarity on what problems interns and visitors would address even before they visit. However, our colleagues had substantial room for creativity in the approaches used to solve these problems. We think that such a balance is crucial. Most of our work with interns and visitors turned into conference papers in premier conferences.
- *Maintain Continuity and Ownership.* Interns and visitors can write code but then they leave! Someone has to maintain continuity of the research project going. We had to spend several months consolidating code written by interns after every summer, taking ownership of it, and providing continuity for the project.
- *Reflect and Assess.* In a research project that spans several years, it is important to regularly reassess the progress you are making towards your main goal. In the SLAM project we did several things that were interesting technically (for example, checking concurrency properties with counting abstractions, heap-logics, etc.) but in the end did not contribute substantially to our main goal of checking device driver rules. We reassessed and abandoned further work on such sub-projects. Deciding what to drop is very important, otherwise one would have too many things to do, and it would be hard to achieve anything.
- *Avoid the Root of All Evil.* It is important not to optimize prematurely. We believe it is best to let the problem space dictate what you will optimize. For example, we used a simple greedy heuristic in NEWTON to pick relevant predicates and we have not needed to change it to date! We also had the experience of implementing complicated optimizations that we thought would be beneficial but were hard to implement and were eventually abandoned because they did not produce substantial improvements.

- *Balance Theory and Practice.* In hindsight, we should have more carefully considered the interactions of pointers and procedures in the SLAM process, as this became a major source of difficulty for us later on (see Section 4.5). Our initial technical report helped us get started and get our interns going, but many difficult problems were left unsolved and unimagined because we did not think carefully about pointers and procedures.
- *Ask for Help.* One should never hesitate to ask for help, particularly if it is possible to get help. With SLAM/SDV, in retrospect, we wish we had asked for help on testing resources sooner.
- *Put Yourself in Another’s Shoes.* Nothing really helped us to prepare for how the product teams operate, how they allocate resources, and how they make decisions. One person’s bureaucracy is another’s structure. Companies with research labs need to help researchers understand how to make use of that structure. On the other hand, researchers have to make a good faith effort to understand how product teams operate and learn about what it takes to turn a prototype into a product.

At this point, SLAM has a future as an analysis engine for SDV. Current research that we are doing addresses limitations of SLAM, such as dealing with concurrency, more accurately reasoning about data structures, and scaling the analysis via compositional techniques. We also want to question the key assumptions we made in SLAM, such as the choice of the Boolean program model. We also hope that the SLAM infrastructure will be used to solve other problems. For example, Shaz Qadeer is using SLAM to find races in multi-threaded programs.

Beyond SLAM and SDV, we predict that in the next five years we will see partial specifications and associated checking tools widely used within the software industry. These tools and methodologies eventually will be integrated with widely used programming languages and environments. Additionally, for critical software domains, companies will invest in software modeling and verification teams to ensure that software meets a high reliability bar.

Acknowledgements. We wish to thank everyone mentioned in this paper for their efforts on the SLAM and SDV projects, and to the many unnamed researchers and developers whose work we built on.

References

- [ABD⁺02] S. Adams, T. Ball, M. Das, S. Lerner, S. K. Rajamani, M. Seigle, and W. Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *SAS 02: Static Analysis Symposium*, LNCS 2477, pages 230–246. Springer-Verlag, 2002.
- [BCDR04] T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, To appear in LNCS. Springer-Verlag, 2004.

- [BCLZ04] T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. *Under review*, 2004.
- [BCM⁺92] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.
- [BCR01] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031. Springer-Verlag, 2001.
- [BLX04] T. Ball, V. Levin, and F. Xei. Automatic creation of environment models via training. In *TACAS 04: Tools and Algorithms for the Construction and Analysis of Systems*, To appear in LNCS. Springer-Verlag, 2004.
- [BMMR01] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.
- [BMR01] T. Ball, T. Millstein, and S. K. Rajamani. Polymorphic predicate abstraction. Technical Report MSR-TR-2001-10, Microsoft Research, 2001.
- [BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. In *POPL 03: Principles of programming languages*, pages 97–105. ACM, 2003.
- [BPR01] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstractions for model checking C programs. In *TACAS 01: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 268–283. Springer-Verlag, 2001.
- [BPR02] T. Ball, A. Podelski, and S. K. Rajamani. On the relative completeness of abstraction refinement. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2280, pages 158–172. Springer-Verlag, April 2002.
- [BPS00] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software-Practice and Experience*, 30(7):775–802, June 2000.
- [BR00a] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.
- [BR00b] T. Ball and S. K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, January 2000.
- [BR01a] T. Ball and S. K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *PASTE 01: Workshop on Program Analysis for Software Tools and Engineering*, pages 97–103. ACM, 2001.
- [BR01b] T. Ball and S. K. Rajamani. SLIC: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research, 2001.
- [BR02a] T. Ball and S. K. Rajamani. Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, January 2002.
- [BR02b] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, January 2002.
- [Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

- [CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In *POPL 77: Principles of Programming Languages*, pages 238–252. ACM, 1977.
- [CCG⁺03] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE 03: International Conference on Software Engineering*, pages 385–395. ACM, 2003.
- [CMP] E. Chailloux, P. Manoury, and B. Pagano. *Développement d’Applications Avec Objective CAML*. O’Reilly (Paris).
- [CYC⁺01] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP 01: Symposium on Operating System Principles*, pages 73–88. ACM, 2001.
- [Das00] M. Das. Unification-based pointer analysis with directional assignments. In *PLDI 00: Programming Language Design and Implementation*, pages 35–46. ACM, 2000.
- [DF01] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI 01: Programming Language Design and Implementation*, pages 59–69. ACM, 2001.
- [DP04] R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Technical Report MSR-TR-2004-07, Microsoft Research, 2004.
- [DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI 02: Programming Language Design and Implementation*, pages 57–68. ACM, June 2002.
- [DNS03] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, 2003.
- [ES01] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *CAV 01: Computer Aided Verification*, LNCS 2102, pages 324–336. Springer-Verlag, 2001.
- [GS97] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Computer-aided Verification*, LNCS 1254, pages 72–83. Springer-Verlag, 1997.
- [HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL ’02*, pages 58–70. ACM, January 2002.
- [Kur94] R.P. Kurshan. *Computer-aided Verification of Coordinating Processes*. Princeton University Press, 1994.
- [LBD⁺04] J. R. Larus, T. Ball, M. Das, Rob DeLine, M. Fähndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy. Righting software. *IEEE Software (to appear)*, 2004.
- [Lei03] K. R. M. Leino. A sat characterization of boolean-program correctness. In *SPIN 03: SPIN Workshop*, LNCS 2648, pages 104–120. Springer-Verlag, 2003.
- [RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.
- [Som98] F. Somenzi. Colorado university decision diagram package. Technical Report available from <ftp://vlsi.colorado.edu/pub>, University of Colorado, Boulder, 1998.
- [SP81] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, pages 189–233. Prentice-Hall, 1981.

Design Verification for Control Engineering

Richard J. Boulton, Hanne Gottliebsen¹, Ruth Hardy², Tom Kelsey², and Ursula Martin¹

¹ Queen Mary University of London,
Ursula.Martin@dcs.qmul.ac.uk

² University of St Andrews

Abstract. We introduce control engineering as a new domain of application for formal methods. We discuss design verification, drawing attention to the role played by diagrammatic evaluation criteria involving numeric plots of a design, such as Nichols and Bode plots. We show that symbolic computation and computational logic can be used to discharge these criteria and provide symbolic, automated, and very general alternatives to these standard numeric tests. We illustrate our work with reference to a standard reference model drawn from military avionics.

1 Introduction

To control an object means to influence its behaviour so as to achieve a desired goal. Control systems may be natural mechanisms, such as cellular regulation of genes and proteins by the gene control circuitry in DNA. They may be man-made – an early mechanical example was Watt’s steam governor – but today most man-made control systems are digital, for example fighter aircraft or CD drives. In genomics we want to identify the control mechanism from observations of its properties: in engineering we want to solve the dual problem of constructing a system with certain properties. Traditionally, control is treated as a mathematical phenomenon, modelled by continuous or discrete dynamical systems. Numerical computation is used to test and simulate these models, for example Matlab is an industry standard in avionics. A largely separate process is then used in the implementation of such as continuous model as a digital (discrete) controller.

Block diagrams are a standard engineering representation of dynamical systems, obtained from a Laplace transform. In a recent paper [3] we added assertions about phase and gain of a signal to block diagrams and devised and implemented a simple Hoare logic. We were able to emulate mechanically engineers informal reasoning about phase and gain. We also prototyped symbolic, automated, very general alternatives to some standard numeric tests used in engineering design, and it is this work which forms the main result of this paper. We replace numeric plots with symbol manipulation in the computer algebra system Maple and the theorem prover PVS. This in turn exploited our Maple-PVS system [12], and Gottliebsen’s PVS continuity checker [13].

Control engineering is a large subject: we intend to focus on those aspects which are to a control engineer fairly standard and widely used in practice [25].

Optimal control assumes that a model of the system is available and one wants to optimise its behaviour, using the calculus of variations and so forth: for example pre-computing a desired flight-path for a spacecraft. Feedback control compensates for uncertainty in the model by using feedback to correct for deviations for desired behaviour: for example if the spacecraft strays off course. Models vary according to the application: for example differential equations are used when modelling a continuous signal, but these are replaced by difference equations when modelling a sampled signal as used in digital systems. In reasoning about such systems we are interested not only in the solutions, but in their properties. These include the time response, stability, frequency response and behaviour under perturbation. The time response considers features such as the time taken for a property of the system (e.g. the cruising speed of a car) to reach the desired value, and by how far it overshoots before settling at the desired value. Stability analysis considers whether the system will always settle into a steady state following a change to the input(s). An output of an unstable system may increase out of control or oscillate. Frequency response considers the amplitude and phase shift of the output signal when the system is presented with a sinusoidal input. The analysis considers input signals with a range of frequencies.

In practice systems are rarely linear: non-linear systems are generally treated locally by linearising at points of interest, but global behaviour is subject of much research and raises subtle questions in differential and algebraic geometry.

In classical control a Laplace transform is applied to a linear system to obtain a representation as a transfer function, a rational function over the complexes. Analysis of properties, such as frequency and response of the control system, is in terms of the position of its roots and poles in the complex plane. So called modern control considers a state-space representation, which replaces a single differential equation with a system of simultaneous equations in the state variables, and analyses the system via properties of the eigenvalues of a related matrix. Both frameworks can be extended from SISO (single input) to MIMO (multiple input) systems.

Block diagrams are often used to represent systems with feedback graphically, for example in classical control a block diagram is a directed graph whose edges are labelled by rational functions over the complexes. They also allow more general representation of components described only by their input/output behaviour.

Software such as the widely used Mathworks Simulink [21], the industry standard in avionics and automotive applications, supports numerical tests and simulations. A number of standard tests are used for prediction and analysis: for example the Nichols plot is a numerical test which investigates stability. It displays the steady state behaviour of a classical control system in terms of the phase and gain of a sinusoidal input. We shall see below how some of the control requirements of fighter aircraft are specified in terms of acceptable paths in this plot [26].

In practice man-made control systems are typically digital embedded software systems, which use sampled, rather than continuous time. These can be

modelled as discrete dynamical systems (difference equations), which again admit a transform representation via the z-transform, and an analogous state-space representation, investigated as before using matrix algebra. The design of a digital controller, for example in avionics applications, typically involves analysis as above in continuous time: it is then passed to a software team for implementation as a discrete digital system. It has been suggested that this process is a likely source of error: indeed apparently “similar” continuous and discrete systems may have very different stability properties. The ubiquity of such embedded controllers, for example in cars and domestic appliances, has led to increased interest in methods of generating assured code straight from a high level design [2].

The study of control in the context of computer science is an emerging area: we identify some strands of work which complement our own:

- The most well developed is the field of hybrid systems, which models certain control systems as automata with discrete transitions which are then amenable to model checking [20] and theoretical analysis [18]. Alur and Dill [5] introduced timed automata, state-transition diagrams annotated with timing constraints using finitely many real-valued clock variables which can be used to model discrete dynamical systems
- In the 1970s Arbib and Manes [1] studied categorical models of linear control: more recently various categories with feedback have been much studied, especially traced monoidal [16] categories, which are models for linear logic. These seem to obey similar algebraic laws to feedback diagrams, though as far as we know the connection has not been developed formally. Tourlas [14] has studied reasoning about general diagram languages.
- Less attention has been paid to the classical dynamical systems representations. Perhaps the closest foundational work is Edalat’s [9] extension of classical domain theory to analysis and dynamical systems. Tiwari [29] allows abstraction of dynamical systems to a level where model checking can be used.
- Our own work on light formal methods for mathematical systems [6,7] was a precursor to this work: we devised an assertion language and lint-like checker for NAG Ltd’s AXIOM system.
- The widespread use of Simulink suggests that effective formal verification techniques for block diagrams could have significant impact. The Clawz system of Arthan et al [2], developed for Qinetiq, is a first step: it translates discrete-time models, described using Simulink, into formal specifications in Z. A controller implementation in an Ada-like programming language can then be verified against these Z specifications using the ProofPower mechanised proof assistant. Qinetiq used Clawz in a case study of the braking systems of the Eurofighter, and are addressing issues of concurrency in this framework using CSP/FDR. Mahony [23] has used similar ideas in his DOVE system.

In a pilot study [3] we developed a Hoare-style logic and a verification condition generator for a restricted class of block diagrams, essentially those with

a tree structure. Hoare logics [17] were originally studied by Hoare, Floyd and others to give an axiomatic basis for programming, and continue to be used for a variety of applications, for example Java byte-code verification [24]. As far as we know our work is the first to investigate Hoare-style logics for feedback systems. We attached assertions to nodes in the diagram: the key observation was that phase and gain were compositional, and hence we could reason about them locally, and propagate our reasoning through the diagram to deduce properties of a classical frequency-response analysis. Following Gordon's approach [11] the logic was mechanised in the HOL98 theorem proving system, allowing goal-directed reasoning, machine assistance in the details of the proof, and automatic generation of verification conditions, the logical formulas that must ultimately be proved to justify an assertion in the Hoare logic. The verification conditions themselves are pure predicate logic formulas, that is, they do not involve the constructs of our logic.

2 Design Requirements for Aerospace Applications

As one might expect the design and verification of flight control laws, and their implementation in flight control systems of civilian or military aircraft, is a mature technology involving many specialist teams in engineering companies and certification authorities - Pratt [26] provides a comprehensive account. Producing a new plane typically takes around three years from the official program start to the first test flight, and can cost upwards of a billion dollars. The customer for a civilian aircraft might expect to get their first plane about a year after the first test flight: the flight test programme for military aircraft may be significantly longer as they are more likely to contain novel technologies to give enhanced performance, and these need to be tested across a greater range of highly dynamic missions, a wider flight envelope and varying payloads. The flight control algorithms represent between five and ten per cent of the flight control law (FCL) software on an aircraft, and this in turn represents around a quarter of the activity of the flight control system (FCS): the rest is associated with safety monitoring, redundancy management and so on. Typically development of the FCL and FCS follows the "V-model", moving down through the specification of the aircraft, system, equipment and components to manufacturing and coding and then up through an integration path involving suitable validation/verification at each level. Design engineers at the component level initially produce a control law for a (continuous) model of the airplane to meet certain design requirements: this is subsequently implemented as a digital controller and then incorporated in the design and test cycle for the full FCS.

The controller is designed to obtain certain desired behaviours of the controlled system: for example we might want it to be stable, and to behave in a specified way, to within an error bound, in response to a given input function, such as a step function, or a sine wave. The desired behaviours constitute design requirements. For larger systems these may be expressed in two parts: a high level description of a property, for example "good flying qualities", and a corresponding evaluation criterion, expressed as a precise mathematical property that

the controlled system is required to have to discharge the design requirement. It is these properties that are often expressed as showing that a particular curve avoids a particular region, and discharged by plotting a graph.

As an example we consider the Garteur HIRM (High Incidence Research Model) benchmark [10]. This was produced in the early nineties as a challenge problem in robust control: the design problem was to produce a control augmentation system for HIRM, an aircraft typical of modern combat aircraft. In general terms the aim of the flight control system is to give good handling qualities across the specified flight envelope and also provide robustness to unmodelled plant dynamics, modelling uncertainties and variations in operating point within the flight envelope. Acceptable noise and disturbance rejection must also be demonstrated.

In general simple approximate models are chosen so as to enhance understanding of the principles of the design. The aircraft definition provides a continuous model of the aircraft dynamics in terms of the flight conditions (which describe things such as aeroplane geometry, altitude, mass number etc), and the response of the aircraft to control inputs (in terms of displacement, velocity and acceleration). Models are also needed of atmospheric conditions and so forth. Thus for example HIRM is described in terms of 11 inputs (relating to taileron, canard and rudder deflections, throttle and wind), 16 states (velocities, roll pitch and yaw rates and angles, centre of gravity¹ etc) and 20 outputs.

Initial design of the flight control laws is carried out against this reference model, to meet the high level design requirements. Pratt [26] provides a thorough account of many of these for modern fighter aircraft. We give two just examples to indicate some of the subtleties of design of modern fighter aircraft:

- Good handling qualities are those that offer precise control in the various modes of operation of the aircraft, with low pilot workload. While the former can be estimated by from the model or judged by experiment, for the latter evaluation by pilots is typically used, and then correlated with parameters in aircraft response which the pilot uses in performing the task. Thus for example the phugoid is a low-frequency mode which causes oscillations in pitch and speed: this can be controlled by pilots but requires attention, so poor phugoid characteristics make for poor handling. Pilot induced oscillations are unwanted oscillations resulting from the pilots attempt to control the aircraft, and require subtle analysis particularly of unexpected behaviours of fly-by-wire systems.
- **aeroservoelasticity** or structural coupling is the name given to the interaction between the flight control system of an aircraft and the oscillations of the airframe. The sensors of the flight control system detect not only the motion of the aircraft, which provides the required feedback to the control system, but also high frequency oscillations due to resonances of the airframe, which can feed into the control loop and cause instability. To solve this problem notch filters are introduced to attenuate the resonances. However these can also add a phase lag, which itself interferes with stability. Correcting the

¹ The centre of gravity may change as the plane burns fuel and “releases items”

phase lag with a phase advance filter introduces further coupling at high frequencies. Pratt [26] provides a detailed survey of how this design conflict was resolved for the Eurofighter, and the techniques used in the evaluation criteria.

Notice that both these problems are made more challenging by the use of more complex automated control systems, lighter more flexible materials in the airframe, and the need to take account of a variety of payloads.

For the HIRM model the design requirements comprise:

- Control strategy
- Pilot commands
- Robustness considerations for the design envelope, modelling, measurement and hardware implementation
- Robustness requirements
- Performance requirements
- Scheduling considerations

and each requirement comes with a corresponding evaluation criterion. The evaluation criteria comprise several hundred numeric tests in the form of response to specified inputs, generally expressed graphically. We expand below on one such test, the Nichols plot, which occurs in about half the criteria.

3 Diagrams as Evaluation Criteria

The Laplace transform provides an algebraic representation of a linear dynamical system as a transfer function $G(s)$, that is a quotient of two polynomials in a complex variable s . Under this representation the input $X(s)$ to $G(s)$ is transformed to $Y(s) = G(s)X(s)$. Transfer functions provide the traditional block diagram representation used in Simulink. Thus for example the HIRM rudder actuator has transfer function

$$\frac{1}{1 + 0.0191401s + 0.000192367s^2}.$$

The behaviour under particular inputs is often represented graphically. For simplicity we describe only the “frequency response” analysis of such a system via Nichols plots. If a sine wave is input to a stable system it can be shown that the steady state output will also be a sine wave, of the same frequency ω as the input, but with different phase and amplitude. The difference between the output and input phases is called the “phase margin”, $\arg(G(I\omega)) = \arg(Y(I\omega)/X(I\omega))$, and ratio of the output to the input amplitudes is called the “gain margin”, that is $|G(I\omega)| = |Y(I\omega)|/|X(I\omega)|$.

Thus a simple evaluation criterion for a design requirement might take the form

- Phase margin greater than 40 degrees
- Gain margin of G greater than 20 dB

The Nichols plot of G allows us to express more complex design requirements. It plots the phase against the gain in decibels for different values of the input frequency ω , and is given parametrically by

$$x = \arg(G(I\omega t)) \quad (1)$$

$$y = 20\ln(|G(I\omega t)|)/\ln(10). \quad (2)$$

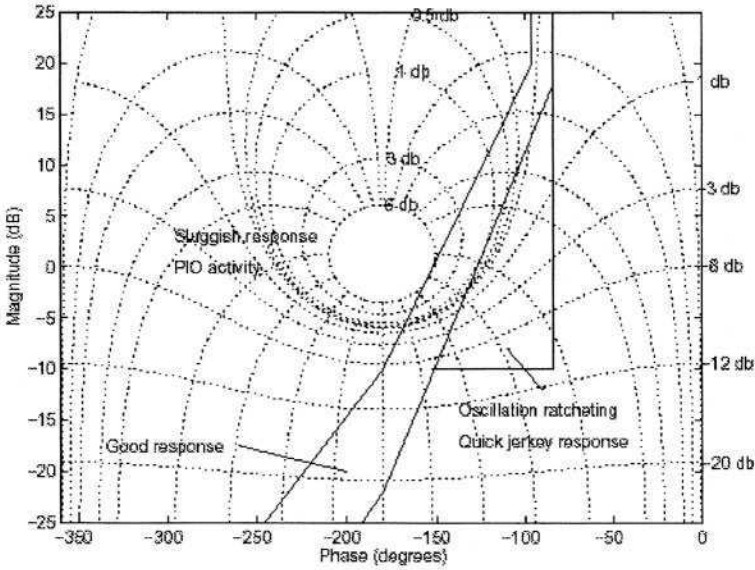


Fig. 1. Handling requirements for HIRM model

Thus one design requirement for the HIRM benchmark is a lengthy description of “acceptable handling qualities” that balance aircraft performance against pilot discomfort, and avoid pilot induced oscillations. The evaluation criteria are that a sequence of Nichols plots all lie in the region marked “good response” in Fig. 1. Other evaluation criteria used in HIRM require that various Nichols plots avoid a hexagonal exclusion region around $(-\pi, 0)$.

Other plots used in evaluation criteria include Bode and Nyquist plots in the frequency domain, and analysis of the response to ramp and step inputs in the time domain. Matlab [22] commands such as `nicols`, `bode` draw the respective plots of a given transfer function, and for small examples GUI tools allow the user to manipulate the plot to a required form, and obtain a modified form of the input that generates this output.

4 Computing with Real Numbers

The methods above require us to discharge verification requirements which take the form of showing that a particular curve in $\mathbf{R} \times \mathbf{R}$ lies in a particular region bounded by straight lines, or more generally that a function is positive or continuous in an interval. We briefly summarise the relevant methods from numeric and symbolic computation, and computational logic.

Numerical methods. Numerical methods are the standard, and almost universal, approach to computational support for analysis of control and dynamical systems, and in particular the discharge of such requirements.

These are widely available through standard commercial libraries such as NAG and MatLab which generate numeric or graphical output, from which various properties of the system may be inferred. In addition such systems can readily accommodate other inputs, for example from measurement devices, or other numerical procedures, such as curve fitting. For many problems, for example the investigation of chaotic phenomena, there are no alternative standard techniques. The main advantage of numerical systems is that they will always give an answer, and with sufficient user expertise are accepted as doing so sufficiently quickly and accurately, with established protocols for testing and error analysis. However the output, and properties derived from it, will be always be numeric and not analytic, and support for investigating other properties of the solution, for example continuity or behaviour under parameters, may be limited. Thus verifying a Nichols plot for a parameterised curve will involve drawing a series of plots for different values of the parameter.

Symbolic computation. Symbolic computation systems, such as Maple, contain a variety of symbolic algorithms that might in principle be used discharge our verification requirements.

We note that we cannot hope for a fully automatic test for an arbitrary function q to be positive in an interval $[a, b]$, as this is a version of the zero-constant problem and hence undecidable, even for q an expression generated by a variable x , the *sin*, *exp* and *modulus* functions [28]. There is much theoretical work on techniques for analysing questions of this kind for particular q , for example quantifier elimination, which is decidable for polynomial functions q , but can still be doubly exponential, and more effective methods such as cylindrical algebraic decomposition, CAD, which still become infeasible for complicated inequalities. Jirstrand [18] experimented with CAD for polynomial differential equations with constraints, and solved some problems involving stationarity, stability, and curve following, but was only able to tackle small examples.

Computer algebra systems test for analytic properties such as continuity, convergence or differentiability by using numeric or symbolic root-finding algorithms to find possible points of failure of the required property, which again reduces to the zero constant problem. While in principle parameters can be handled, computer algebra systems do not generally handle pre and side conditions well []. Consider the equation

$$y'(x) + y/(2\sqrt{x-a}) = \ln(b-x) \exp(-\sqrt{x-a}) \quad (3)$$

for which Maple returns

$$y(x) = \exp(-\sqrt{x-a})((x-b)(\ln(b-x)-1) + C) \quad (4)$$

This supposed solution is only defined if $x > a$ and $x < b$, so that if $b < a$ it is not defined anywhere on the real line. Maple has produced this solution because the `dsolve` procedure, for finding a solution of an equation valid in an interval V , does not check what the interval V might be, or the required conditions that functions are continuous on it.

Computational logic and real number theorem proving. Computational logic means the use of a computer to produce a proof in some formal system. A variety of systems have been built, some motivated by experimentation in a particular formal system, others by the need for increasingly complex practical verification, especially hardware and distributed systems. We mention below other applications of computational logic to verification of control systems. Such systems may be hard to use, but the pay-off is that the user can be confident that the results that they produce are correct. SRI's PVS [27] - prototype verification system - is based on sequent calculus and provides a collection of powerful primitive inference mechanisms including prepositional and quantifier rules, induction, rewriting, and decision procedures for linear arithmetic. The implementations of these mechanisms are optimised for large proofs: there is support for proof strategies and a powerful brute force search mechanism called *grind*. PVS has a rich higher-order type system supporting overloading of operators, subtypes and dependent types, and mechanisms for parametric specifications. It has been widely used in applications, particularly aerospace work.

As we saw in Section 1 computational logic has impacted control engineering: here we concentrate on those aspects useful in the discharge of design requirements.

Some form of real analysis has been implemented in many theorem provers, both because it forms the basis of much mathematics and, increasingly, because of its use in hardware verification: for example Harrison's work on floating point verification [15]. PVS has a basic built-in theory of the reals: the axiomatisation is via the least upper bound principle, every non-empty set of reals bounded above has a least upper bound. Dutertre [8] extended this and developed a library for real analysis in PVS, including definitions and basic properties of convergence, limits, continuity and differentiation. Further development of real analysis in PVS is described in [13], where the transcendental functions were incorporated by defining the functions in terms of power series and constructing on top of the basic definitions a large lemma database of routine results about elementary functions, that is functions built up from rational functions of a variable x together with *cos*, *sin*, *exp* and *log*. Typical entries are:

$$\begin{aligned} \sin\left(\frac{\pi}{2}\right) &= 1 & \cos(\pi) &= -1 \\ \sin(x) &= \cos\left(\frac{\pi}{2} - x\right) & \frac{d}{dx}(\sin(x)) &= \cos(x) \end{aligned}$$

Particularly useful for our work are procedures that attempt to check if a function has a property in a closed interval [13]: typical properties are continuity, convergence and differentiability. The checker relies on results which follow the text-book development of continuity (sums of continuous functions are continuous and so on), augmented with our database and a collection of standard results about the continuity of elementary functions.

The method used for this checking is what one might call the High School method. It is based on the theorems that the constant functions and the identity function are continuous everywhere, and that well-founded combinations using the following operators are also continuous: addition, subtraction, multiplication, division, absolute value and function composition. Also, the functions $\exp$, $\cos$, $\sin$ and $\tan$ are continuous everywhere in their domains. The checker can be used to check the continuity and limiting behaviour of functions such as

$$e^{x^2+|1-x|} \text{ or } -\pi - 1 + \pi * e^{1-\cos(x)}.$$

As continuity is undecidable [4], this will not always work, however for many examples it is sufficient. Note also that if a proof fails using the checker we can always go back and prove the result from first principles and add it to the database. The evolving nature of the database makes it hard to provide efficiency measures. At first sight it might appear that using a computational logic engine for such symbolic analysis gives one no advantage over a computer algebra system, and the considerable disadvantage that it is very much harder to use. However as we have seen computer algebra systems have a number of disadvantages, and computational logic engines like PVS have the advantage that the results they produce are correct and unambiguous. Our methods may fail if the lemma database does not contain an appropriate result: however if they report success this will always be justified by a proof, rather than, as in a computer algebra system, failure of a root finding algorithm or similar.

Maple-PVS. Our Maple-PVS system [12] combines symbolic computation and computational logic by providing restricted invocation of PVS from Maple to perform the checks we describe above. These are called from Maple by a simple pipe-lined interface. As well as direct calls from Maple to verify continuity and so on, we are able to build these checks into other Maple procedures. For example, it is straightforward to write a harness for a Maple routine to solve differential equations which checks the validity of the pre-conditions for the routine to be correct.

5 Symbolic Diagram Testing with Maple-PVS

We described above how classical control engineering gives rise to design requirements expressed geometrically, typically in the form of plots such as the Nichols plots. In this section we show how to discharge these requirements automatically in some cases using symbolic computation and computational logic embedded in our Maple-PVS system.

Our approach is based in the following simple idea. Tests based on such plots concern showing that a given curve $y = f(x)$ is bounded above (or below) by a given line $y = g(x)$ in an interval $[a, b]$, or equivalently that

$$\forall x \in [a, b], g(x) - f(x) > 0. \quad (5)$$

The following two theorems each provide a sufficient condition for the inequality to hold which is readily verified symbolically for certain lines and curves.

Theorem 1. *Given real-valued functions f, g defined throughout an interval $[a, b]$ and satisfying*

1. g is linear in $[a, b]$
2. $f(x)$ is continuous and twice differentiable in $[a, b]$
3. $f(x)$ is monotonic increasing and concave in $[a, b]$, ie $f'(x) > 0$ and $f'(x)$ is monotonically decreasing in $[a, b]$, ie $f'(x) > 0$ and $f''(x) < 0$ in $[a, b]$.
4. $g(b) > f(b)$
5. $g'(b) < f'(b)$

then $f(x) < g(x)$ throughout the region $[a, b]$.

The proof of this is straightforward. Let $g(x) = px + q$. The continuous function $g(x) - f(x)$ attains its maximum and minimum values in an interval at endpoints or at zeroes of $p - f'(x)$. Now for any x in $[a, b]$ we have $p - f'(x) < p - f'(b) = g'(b) - f'(b) < 0$, so the minimum value of $g(x) - f(x)$ is either $g(b) - f(b) > 0$, or $g(a) - f(a) \geq g(b) - f(b) + (f'(b) - p)(b - a) > 0$.

Theorem 2. *Given real-valued functions f, g defined throughout an interval $[a, b]$ and satisfying*

1. g is linear in $[a, b]$
2. $f(x)$ is continuous and twice differentiable in $[a, b]$
3. $f(x)$ is monotonic increasing and convex in $[a, b]$, ie $f'(x) > 0$ and $f'(x)$ is monotonically increasing in $[a, b]$, ie $f'(x) > 0$ and $f''(x) > 0$ in $[a, b]$.
4. $g(b) > f(b)$
5. $g(a) > f(a)$

then $f(x) < g(x)$ throughout the region $[a, b]$.

Similar results hold for $f(x)$ monotonic decreasing, and with a and b interchanged.

Theorem 1 ensures the correctness of the following test for condition (1):

1. compute the verification conditions (i)-(v) above
2. verify that conditions (i)-(v) hold

For any $g(x)$ linear and $f(x)$ continuous and twice differentiable in $[a, b]$, we can partition $[a, b]$ into intervals determined by the zeroes and points of inflection of $g(x) - f(x)$ and $g'(x) - f'(x)$, and hence apply one or other of these theorems in each interval to determine whether f is bounded above by g throughout.

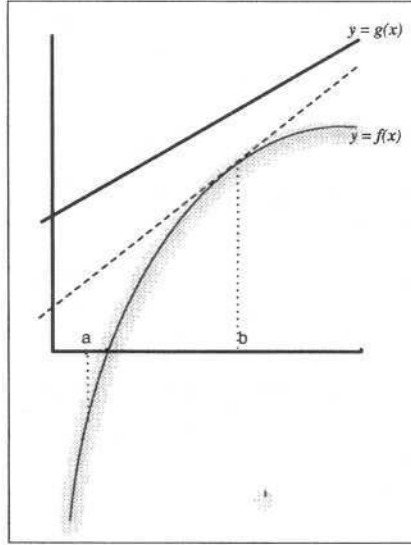


Fig. 2. Conditions of Theorem 1

In the case of the Nichols plot of a transfer function $G(s)$, which is a rational function (ie a quotient of two polynomials), the curve $y = f(x)$ will be given parametrically by

$$x = \arg(G(I\omega t)) \quad (6)$$

$$y = 20 \ln(|G(I\omega t)|) / \ln(10) \quad (7)$$

and its derivatives $f'(x)$, $f''(x)$ will be quotients of polynomials in $\cos$, $\sin$ and x .

We carried out a prototype implementation of the test using our Maple-PVS system described above. Maple readily computes the verification conditions, which require computing and simplifying derivatives. PVS is used to discharge them, using its tests for continuity and inequality of elementary functions.

Example

As an example consider the transfer function

$$G(s) = k.(s^2 + 2cs + d)^{-1} \quad (8)$$

and Nichols plot in the region $[-\pi, -\pi/2]$. This is the graph of $|G(Iw)|$ against $20 \log_{10} |\tan^{-1}(\text{Im}(G(Iw))/\text{Re}(G(Iw)))|$, given parametrically by

$$x := \arg(k.(-t^2 + 2Ict + d)^{-1}) \quad (9)$$

$$y := (20 \ln |k.(-t^2 + 2Ict + d)^{-1}|) / \ln(10). \quad (10)$$

Then, as an example, Condition (iii) requires us to calculate dy/dx and show positive in $[-\pi, -\pi/2]$. Maple shows

$$dy/dx = 20 \sin(x)(-c Q + c^2 \cos(x) + d^2 \cos(x))/Q/(c \cos(x) + Q/\ln(10)) \quad (11)$$

where

$$Q = (c^2 \cos(x)^2 + d^2 \sin(x)^2)^{(1/2)}. \quad (12)$$

In this case the PVS test that dy/dx is positive in the desired region follows the usual informal human reasoning - parse the expression into a quotient of sums of products, use the lemma database to identify the sign of each component - for example $\sin$ and $\cos$ are both negative in the given interval - and hence deduce the sign of the products, then the sums, then the quotient.

It is hard to provide a complexity or timing analysis of our method, since it is built on top of an evolving lemma database. For the kind of applications we consider the inputs are not random, but typically consist of a large number of tests in rather similar format - for example Nichols plots are usually computed over the interval $[-\pi, -\pi/2]$, and in the HIRM model nearly all satisfy the conditions of Theorem 1. Hence it is worthwhile optimising the system by extending the lemma database as required. Our method is related to what is called, in computer graphics, the method of Lipschitz bounds, devised by Kalra and Barr [19] to handle ray-tracing as a technique for visualising implicit surfaces.

6 Analysis

We are aiming to find automated symbolic methods of design testing, to replace the use of numerical graph plots which are verified by inspection.

We prototyped our method for Theorem 1 and another similar test, and performed the necessary computations in Maple and PVS. Maple can readily provide the support that we need to compute derivatives, simplify and so on. PVS contains definitions and properties of elementary functions and likewise provides much of what we need, and the necessary material is under constant development, particularly in support of NASA's verification work on air traffic control algorithms.

There are a number of other symbolic computation and theorem proving technologies which could be used to implement this approach to design testing - in particular it would be fairly straightforward to eliminate the dependency on Maple altogether and use a rewrite engine to compute and simplify derivatives. Longer term there is no obstacle in principle to embedding the whole process as an automated call from other engines, like Simulink.

Note that unlike numeric tests our methods are not samples, but provide verification, subject to the correctness of the underlying Maple-PVS implementation, for all values in the interval. We hope to be able to use similar methods for robust control, where we will need to consider parameterised curves, and to extend to non-linear systems. Current techniques like Bode and Nichols plots are

a heritage of the pre-computer days when they really were plotted on graph paper. However the availability of methods like ours encourages speculation about other design verification methods which while impossible to plot and eyeball may be susceptible to symbolic verification.

Acknowledgements. This research supported by QinetiQ, EPSRC grants GR/M98340 and GR/L48256, and by an EPSRC studentship to the third author. We are indebted to Manuela Bujorianu, John Hall, Rick Hyde, and Yoge Patel for their insights into control engineering, and to Rob Arthan and Colin O'Halloran for helpful discussions.

References

1. M Arbib and E Manes Machines in a category SIAM review 57 (1974), 163-192
2. R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *Proc. 3rd IEEE International Conference on Formal Engineering Methods (ICFEM 2000)*, York, September 2000.
3. Richard J. Boulton, Ruth Hardy and Ursula Martin A Hoare Logic for Single-Input Single-Output Continuous-Time Control Systems In *Proceedings 6th International Workshop on Hybrid Systems, Computation and Control*, pages 113-125, Springer LNCS vol 2623, 2003
4. G Cherlin Rings of continuous functions: decision problems, In *Model theory of algebra and arithmetic*, Lecture Notes in Mathematics 834, pages 44–91, Springer 1980
5. D Dill A theory of timed automata Theoretical Computer Science , 126, pages 183-235,1994
6. Martin Dunstan, Tom Kelsey, Ursula Martin and Steve Linton Lightweight formal methods for computer algebra systems In *ISSAC'98: Proc ACM International Symposium on Symbolic and Algebraic Computation*, Rostock, ACM Press, 1998
7. Ursula Martin, Martin Dunstan, Tom Kelsey and Steve Linton Formal methods for extensions to computer algebra systems In *Proc FM'99: World Congress on Methods in the design of computing systems* Lecture Notes in Computer Science 1709, pages 1758-1777, Springer Verlag, 1999
8. B. Dutertre Elements of Mathematical Analysis in PVS, In *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs '96*, Lecture Notes in Computer Science 1125, pages 141-156, Springer-Verlag 1996
9. A Edalat and A Lieutier Domain theory and differential calculus Proc IEEE LICS 17, IEEE Press 2002
10. Robust Flight Control Design Challenge Problem Formulation and Manual: the High Incidence Research Model (HIRM) Garteur - Group for aeronautical research and technology in Europe Technical report, GARTEUR/TP-088-4, 1997
11. M. J. C. Gordon. Mechanizing programming logics in higher order logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 387–439. Springer-Verlag, 1989.
12. Hanne Gottlieb, Tom Kelsey and Ursula Martin Hidden verification for computer algebra systems To appear, Journal of Symbolic Computation 2004
13. Hanne Gottlieb Transcendental Functions and Continuity Checking in PVS In *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000* Lecture Notes in Computer Science 1869, pages 198-215, Springer-Verlag 2000

14. C. Gurr and K. Tourlas. Towards the principled design of software engineering diagrams. In *Proc. 22nd International Conference on Software Engineering*, pages 509–520, ACM Press, 2000.
15. J Harrison. Theorem proving in the real numbers. Cambridge University Press, 1995.
16. M. Hasegawa. *Models of Sharing Graphs*. Springer, 1997.
17. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 583, 1969.
18. Mats Jirstrand. Nonlinear control system design by quantifier elimination, *J. Symbolic Comput.*, 24(1997), pp 137–152.
19. D. Kalra and A.H. Barr. Guaranteed Ray Intersections with Implicit Surfaces. *Computer Graphics (SIGGRAPH '89 Proceedings)*, Vol. 23(3), 1989, pages 297–306.
20. Bruce Krogh. Approximating Hybrid System Dynamics for Analysis and Control. *HSCC 1999*, LNCS 1569, Springer, 1999.
21. The MathWorks. Simulink. <http://www.mathworks.com/products/simulink/>
22. The MathWorks. Matlab. <http://www.mathworks.com/products/matlab/>
23. B. Mahony. The DOVE approach to the design of complex dynamic processes. In *Proc. of the First International Workshop on Formalising Continuous Mathematics*, NASA conference publication NASA/CP-2002-211736, pages 167–187, 2002.
24. T. Nipkow. Hoare Logics in Isabelle/HOL. In *Proof and System-Reliability*, pages 341–367, Kluwer, 2002.
25. K. Ogata. *Modern Control Engineering*. Prentice-Hall, third edition, 1997.
26. R. W. Pratt, editor. *Flight Control Systems: Practical Issues in Design and Implementation*, volume 57 of *IEE Control Engineering Series*. The Institution of Electrical Engineers, 2000.
27. S. Owre, J. Rushby, and N. Shankar. PVS: a prototype verification system. In *11th Conf. on Automated Deduction*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Verlag, 1992.
28. D Richardson. Some Unsolvable Problems Involving Elementary Functions of a Real Variable. *J. Symbolic Logic* 33, 514–520, 1968.
29. A. Tiwari and G. Khanna. Series of abstractions for hybrid automata. In *Proc. 5th International Workshop on Hybrid Systems: Computation and Control (HSCC 2002)*, volume 2289 of *Lecture Notes in Computer Science*, Springer, 2002.

Integrating Model Checking and Theorem Proving in a Reflective Functional Language

Tom Melham

Oxford University Computing Laboratory
Wolfson Building, Parks Road
Oxford, OX1 3QD, England
Tom.Melham@comlab.ox.ac.uk

Abstract. *Forte* is a formal verification system developed by Intel's Strategic CAD Labs for applications in hardware design and verification. Forte integrates model checking and theorem proving within a functional programming language, which both serves as an extensible specification language and allows the system to be scripted and customized. The latest version of this language, called *reFlect*, has quotation and antiquotation constructs that build and decompose expressions in the language itself. This provides combination of pattern-matching and reflection features tailored especially for the Forte approach to verification. This short paper is an abstract of an invited presentation given at the International Conference on Integrated Formal Methods in 2004, in which the philosophy and architecture of the Forte system are described and an account is given of the role of *reFlect* in the system.

1 The Forte Verification Environment

Forte [17] is a formal verification environment that has been very effective on large-scale, industrial hardware verification problems at Intel [10,11,12,15]. The Forte system combines several model checking and decision algorithms with lightweight theorem proving in higher-order logic. These reasoning tools are tightly integrated within a strongly-typed, higher-order functional programming language called FL. This allows the Forte environment to be customised and large proof efforts to be organized and scripted effectively. FL also serves as an expressive language for specifying hardware behaviour.

Model checking using symbolic trajectory evaluation ('STE') lies at the core of the Forte environment. STE [16] can be viewed as a hybrid between a symbolic simulator and a symbolic model checker. As a simulator, STE can compute symbolic expressions giving outputs as a function of arbitrary inputs. As a model checker, it can automatically check the validity of a simple temporal logic formula—computing an exact characterization of the region of disagreement if the formula is not unconditionally satisfied. These features provide a seamless connection between simulation and verification as well as excellent feedback on failed proof attempts—two key elements of an effective usage methodology for large-scale formal verification [10,17].

STE is a particularly efficient model checking algorithm, in part because it has a very restricted temporal logic. But STE, like any model checker, still has very limited capacity. Forte therefore complements STE with a higher-order logic theorem prover of similar design to the HOL system [6]. Theorem proving bridges the gap between big, practically-important verification tasks and tractable model checking problems. The Forte philosophy is to have as thin a layer of theorem proving as possible, since using this technology is still difficult. But case studies have shown that a surprising amount of added value can be gained from even very simple (mathematically ‘shallow’) theorem proving.

The Forte approach is to tightly integrate model checking and theorem proving within the single framework of a functional programming language and its runtime system. A highly engineered implementation of STE is built into the core of the language, with many entry points provided as user-visible functions. Two key aspects of this architecture are that it is a ‘white-box’ integration of model checking and theorem proving and that functional programming plays a central role in scripting verification efforts.

2 The *reFlect* Functional Language

The successor to FL for future generations of Forte is a new functional language called *reFlect* [7]. The *reFlect* language is strongly typed and similar to ML [8], but has quotation and antiquotation constructs like those in LISP but in a typed setting. This provides combination of pattern-matching and reflection tailored especially for the Forte approach to verification. In what follows, a brief sketch is given of the motivation for the design of these features.

In higher-order logic theorem provers like HOL the logical ‘object language’ in which reasoning is done is embedded as a data-type in the (functional) meta-language used to control the reasoning. This makes the various term analysis and transformation functions required by a theorem prover straightforward to implement. But separating the object-language and meta-language also causes duplication and inefficiency. Many theorem provers, for example, need to include special code for efficient execution of object-language expressions [2,3].

In *reFlect*, the data-structure used by the underlying language implementation to represent syntax trees is made available as a data-type within the language itself. Functions on that data-structure, such as evaluation, are also made available. This approach retains all the term inspection and manipulation abilities of a conventional theorem prover while borrowing an efficient execution mechanism from the meta-language implementation.

It also builds *reflection* [9] into the logic of the theorem prover. In systems like HOL, higher order logic is constructed along the lines of Church’s formulation of simple type theory [5], in which the logic is defined on top of the λ -calculus. Defining a logic on top of *reFlect* in the same way gives a higher-order logic that includes the *reFlect* reduction rules as well as certain reflection inference rules.

These reflection capabilities allow Forte to make a logically principled connection between theorems in higher order logic and the results of invoking a model

checker. A similar mechanism called *lifted-FL* [1] was available in earlier versions of Forte, but *reFlect* provides much richer possibilities. For example, one can use quantifiers to create a bookkeeping framework that cleanly separates logical content from model-checking control parameters.

In addition to serving as a meta-language for theorem proving, functional programming languages have often been used to describe the structure of hardware designs. Notable examples include work done in Haskell [4,14] and LISP [13]. A key capability exploited by such work is simulation of hardware designs by program execution. In Forte, however, we also wish to do various operations on the abstract syntax of models written in the language, as well as straight simulation. For example, we wish to implement and possibly even verify circuit design transformations [20]. *reFlect* makes this a built-in part of the language.

The *reFlect* language can be seen as an application-specific contribution to the field of *meta-programming* [18]. Unlike most meta-programming systems, however, the target applications for *reFlect* in Forte give *intensional analysis* a primary role in the language. Its design is therefore somewhat different from staged functional languages like MetaML [21] and Template Haskell [19], which are aimed more at program generation and the control and optimization of evaluation.

Acknowledgements. I thank the organisers of the IFM 2004 for their kind invitation to speak at the conference in Canterbury. The Forte framework [17] and the *reFlect* language [7] are the result of many years of research and development by Intel's Strategic CAD Labs in Portland Oregon. The research reported in this paper was done in collaboration with Jim Grundy and John O'Leary at Intel, and builds on the Forte work of Carl Seger, Robert Jones, and Mark Aagaard.

References

1. M. D. Aagaard, R. B. Jones, and C.-J. H. Seger, 'Lifted-fl: A Pragmatic Implementation of Combined Model Checking and Theorem Proving', in *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs 1999*, edited by Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, LNCS, vol. 1690 (Springer-Verlag, 1999), pp. 23–340.
2. B. Barras, 'Proving and Computing in HOL', in *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, edited by M. Aagaard and J. Harrison, LNCS, vol. 1869 (Springer-Verlag, 2000), pp. 17–37.
3. S. Berghofer and T. Nipkow, 'Executing higher order logic', in *Types for Proofs and Programs: International Workshop, TYPES 2000*, edited by P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, LNCS, vol. 2277 (Springer-Verlag, 2000), pp. 24–40.
4. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, 'Lava: Hardware design in Haskell', in *Functional Programming: International Conference, ICFP 1998*, (ACM Press, 1998), pp. 174–184.
5. A. Church, 'A Formulation of the Simple Theory of Types', *Journal of Symbolic Logic*, vol. 5 (1940), pp. 56–68.
6. M. J. C. Gordon and T. F. Melham (editors), *Introduction to HOL: A theorem proving environment for higher order logic* (Cambridge University Press, 1993).

7. J. Grundy, T. Melham, and J. O’Leary. ‘A Reflective Functional Language for Hardware Design and Theorem Proving’, Research Report PRG-RR-03-16, Programming Research Group, Oxford University (October, 2003).
8. R. Harper, D. MacQueen, and R. Milner, ‘Standard ML’, Report 86-2, University of Edinburgh, Laboratory for Foundations of Computer Science (1986).
9. J. Harrison, ‘Metatheory and Reflection in Theorem Proving: A Survey and Critique’, Technical Report CRC-053, SRI Cambridge (1995).
10. R. B. Jones, J. W. O’Leary, C.-J. H. Seger, M. D. Aagaard, and T. F. Melham, ‘Practical formal verification in microprocessor design’ *IEEE Design & Test of Computers*, vol. 18, no. 4 (July/August, 2001), pp. 16-25.
11. R. Kaivola and K. R. Kohatsu, ‘Proof engineering in the large: Formal verification of the Pentium-4 floating-point divider’, in *Correct Hardware Design and Verification Methods: 11th Advanced Research Working Conference, CHARME 2001*, edited by T. Margaria and T. F. Melham, LNCS, vol. 2144 (Springer-Verlag, 2001), pp. 196–211.
12. R. Kaivola and N. Narasimhan, ‘Formal verification of the Pentium-4 multiplier’, in *High-Level Design Validation and Test: 6th International Workshop, HLDVT 2001* (IEEE Computer Society Press, 2001), pp. 115–122,
13. M. Kaufmann, P. Manolios, and J. S. Moore (editors), *Computer-Aided Reasoning: ACL2 Case Studies*, (Kluwer, 2000).
14. J. Matthews, B. Cook, and J. Launchbury, ‘Microprocessor specification in Hawk’ in *IEEE International Conference on Computer Languages*, (IEEE Computer Society Press, 1998), pp. 90–101.
15. J. O’Leary, X. Zhao, R. Gerth, and C.-J. H. Seger, ‘Formally Verifying IEEE Compliance of Floating-Point Hardware’, *Intel Technical Journal* (First quarter, 1999). Available at developer.intel.com/technology/itj/.
16. C.-J. H. Seger and R. E. Bryant, ‘Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories’, *Formal Methods in System Design*, vol. 6, no. 2 (March 1995), pp. 147-189.
17. C.-J. H. Seger, R. B. Jones, J. W. O’Leary, M. D. Aagaard, C. Barrett, and D. Syme, ‘An Industrially Effective Environment for Formal Hardware Verification’. Submitted for publication.
18. T. Sheard, ‘Accomplishments and Research Challenges in Meta-Programming’, in *Semantics, Applications, and Implementation of Program Generation: 2nd International Workshop, SAIG 2001*, edited by W. Taha, LNCS, vol. 2196 (Springer-Verlag, 2001), pp. 2-44.
19. T. Sheard and S. Peyton Jones, ‘Template Meta-Programming for Haskell’, in *ACM SIGPLAN Haskell Workshop*, edited by M. T. M. Chakravarty, (ACM Press, 2002), pp. 1-16.
20. G. Spirakis, ‘Leading-edge and future design challenges: Is the classical EDA ready?’, in *Design Automation: 40th ACM/IEEE Conference, DAC 2003* (ACM Press, 2003), p. 416.
21. W. Taha and T. Sheard, ‘Multi-stage programming with explicit annotations’, *SIGPLAN Notices*, vol. 32, no. 12 (2002), pp. 203-217.

A Tutorial Introduction to Designs in Unifying Theories of Programming

Jim Woodcock and Ana Cavalcanti

University of Kent
Computing Laboratory
Canterbury UK

{J.C.P.Woodcock,A.L.C.Cavalcanti}@kent.ac.uk

Abstract. In their *Unifying Theories of Programming (UTP)*, Hoare & He use the alphabetised relational calculus to give denotational semantics to a wide variety of constructs taken from different programming paradigms. A key concept in their programme is the *design*: the familiar precondition-postcondition pair that describes the contract between a programmer and a client. We give a tutorial introduction to the theory of alphabetised relations, and its sub-theory of designs. We illustrate the ideas by applying them to theories of imperative programming, including Hoare logic, weakest preconditions, and the refinement calculus.

1 Introduction

The book by Hoare & He [6] sets out a research programme to find a common basis in which to explain a wide variety of programming paradigms: unifying theories of programming (UTP).. Their technique is to isolate important language features, and give them a denotational semantics. This allows different languages and paradigms to be compared.

The semantic model is an alphabetised version of Tarski's relational calculus, presented in a predicative style that is reminiscent of the schema calculus in the Z [14] notation. Each programming construct is formalised as a relation between an initial and an intermediate or final observation. The collection of these relations forms a *theory* of the paradigm being studied, and it contains three essential parts: an alphabet, a signature, and healthiness conditions.

The *alphabet* is a set of variable names that gives the vocabulary for the theory being studied. Names are chosen for any relevant external observations of behaviour. For instance, programming variables x , y , and z would be part of the alphabet. Also, theories for particular programming paradigms require the observation of extra information; some examples are a flag that says whether the program has started (*okay*); the current time (*clock*); the number of available resources (*res*); a trace of the events in the life of the program (*tr*); or a flag that says whether the program is waiting for interaction with its environment (*wait*). The *signature* gives the rules for the syntax for denoting objects of the theory. *Healthiness conditions* identify properties that characterise the theory. Each healthiness condition embodies an important fact about the computational model for the programs being studied.

Example 1 (Healthiness conditions).

1. The variable *clock* gives us an observation of the current time, which moves ever onwards. The predicate *B* specifies this.

$$B \triangleq \text{clock} \leq \text{clock}'$$

If we add *B* to the description of some activity, then the variable *clock* describes the time observed immediately before the activity starts, whereas *clock'* describes the time observed immediately after the activity ends. If we suppose that *P* is a healthy program, then we must have that $P \Rightarrow B$.

2. The variable *okay* is used to record whether or not a program has started. A sensible healthiness condition is that we should not observe a program's behaviour until it has started; such programs satisfy the following equation.

$$P = (\text{okay} \Rightarrow P)$$

If the program has not started, its behaviour is not described. $\square$

Healthiness conditions can often be expressed in terms of a function ϕ that makes a program healthy. There is no point in applying ϕ twice, since we cannot make a healthy program even healthier. Therefore, ϕ must be idempotent: $P = \phi(P)$; this equation characterises the healthiness condition. For example, we can turn the first healthiness condition above into an equivalent equation, $P = P \wedge B$, and then the following function on predicates $\text{and}_B \triangleq \lambda X \bullet P \wedge B$ is the required idempotent.

The relations are used as a semantic model for unified languages of specification and programming. Specifications are distinguished from programs only by the fact that the latter use a restricted signature. As a consequence of this restriction, programs satisfy a richer set of healthiness conditions.

Unconstrained relations are too general to handle the issue of program termination; they need to be restricted by healthiness conditions. The result is the theory of designs, which is the basis for the study of the other programming paradigms in [6]. Here, we present the general relational setting, and the transition to the theory of designs.

In the next section, we present the most general theory of UTP: the alphabetised predicates. In the following section, we establish that this theory is a complete lattice. Section 4 discusses Hoare logic and weakest preconditions. Section 5 restricts the general theory to designs. Next, in Section 6, we present an alternative characterisation of the theory of designs using healthiness conditions. After that, we rework the Hoare logic and weakest preconditions definitions; we also outline a novel formalisation of Morgan's calculus based on designs. Finally, we conclude with a summary and a brief account of related work.

2 The Alphabetised Relational Calculus

The alphabetised relational calculus is similar to Z's schema calculus, except that it is untyped and rather simpler. An *alphabetised predicate* $(P, Q, \dots, \text{true})$ is an

alphabet-predicate pair, where the predicate's free variables are all members of the alphabet. Relations are predicates in which the alphabet is composed of undecorated variables ($x, y, z, \dots$) and dashed variables ($x', a', \dots$); the former represent initial observations, and the latter, observations made at a later intermediate or final point. The alphabet of an alphabetised predicate P is denoted αP , and may be divided into its before-variables ($\text{in}\alpha P$) and its after-variables ($\text{out}\alpha P$). A *homogeneous relation* has $\text{out}\alpha P = \text{in}\alpha P'$, where $\text{in}\alpha P'$ is the set of variables obtained by dashing all variable in the alphabet $\text{in}\alpha P$. A *condition* ($b, c, d, \dots, \text{true}$) has an empty output alphabet.

Standard predicate calculus operators can be used to combine alphabetised predicates. Their definitions, however, have to specify the alphabet of the combined predicate. For instance, the alphabet of a conjunction is the union of the alphabets of its components: $\alpha(P \wedge Q) = \alpha P \cup \alpha Q$. Of course, if a variable is mentioned in the alphabet of both P and Q , then they are both constraining the same variable.

A distinguishing feature of UTP is its concern with program development, and consequently program correctness. A significant achievement is that the notion of program correctness is the same in every paradigm in [6]: in every state, the behaviour of an implementation implies its specification.

If we suppose that $\alpha P = \{a, b, a', b'\}$, then the *universal closure* of P is simply $\forall a, b, a', b' \bullet P$, which is more concisely denoted as $[P]$. The correctness of a program P with respect to a specification S is denoted by $S \sqsubseteq P$ (S is refined by P), and is defined as follows.

$$S \sqsubseteq P \quad \text{iff} \quad [P \Rightarrow S]$$

Example 2 (Refinement). Suppose we have the specification $x' > x \wedge y' = y$, and the implementation $x' = x + 1 \wedge y' = y$. The implementation's correctness is argued as follows.

$$\begin{aligned} x' > x \wedge y' = y &\sqsubseteq x' = x + 1 \wedge y' = y && [\text{definition of } \sqsubseteq] \\ = [x' = x + 1 \wedge y' = y \Rightarrow x' > x \wedge y' = y] && [\text{universal one-point rule, twice}] \\ = [x + 1 > x \wedge y = y] && [\text{arithmetic and reflection}] \\ = \text{true} \end{aligned}$$

And so, the refinement is valid. □

As a first example of the definition of a programming constructor, we consider conditionals. Hoare & He use an infix syntax for the conditional operator, and define it as follows.

$$\begin{aligned} P \triangleleft b \triangleright Q &\hat{=} (b \wedge P) \vee (\neg b \wedge Q) && \text{if } \alpha b \subseteq \alpha P = \alpha Q \\ \alpha(P \triangleleft b \triangleright Q) &\hat{=} \alpha P \end{aligned}$$

Informally, $P \triangleleft b \triangleright Q$ means P if b else Q .

The presentation of conditional as an infix operator allows the formulation of many laws in a helpful way.

L1	$P \triangleleft b \triangleright P = P$	<i>idempotence</i>
L2	$P \triangleleft b \triangleright Q = Q \triangleleft \neg b \triangleright P$	<i>symmetry</i>
L3	$(P \triangleleft b \triangleright Q) \triangleleft c \triangleright R = P \triangleleft b \wedge c \triangleright (Q \triangleleft c \triangleright R)$	<i>associativity</i>
L4	$P \triangleleft b \triangleright (Q \triangleleft c \triangleright R) = (P \triangleleft b \triangleright Q) \triangleleft c \triangleright (P \triangleleft b \triangleright R)$	<i>distributivity</i>
L5	$P \triangleleft \text{true} \triangleright Q = P = Q \triangleleft \text{false} \triangleright P$	<i>unit</i>
L6	$P \triangleleft b \triangleright (Q \triangleleft b \triangleright R) = P \triangleleft b \triangleright R$	<i>unreachable branch</i>
L7	$P \triangleleft b \triangleright (P \triangleleft c \triangleright Q) = P \triangleleft b \vee c \triangleright Q$	<i>disjunction</i>
L8	$(P \odot Q) \triangleleft b \triangleright (R \odot S) = (P \triangleleft b \triangleright R) \odot (Q \triangleleft b \triangleright S)$	<i>interchange</i>

In the Interchange Law (L8), the symbol $\odot$ stands for any truth-functional operator.

For each operator, Hoare & He give a definition followed by a number of algebraic laws as those above. These laws can be proved from the definition. As an example, we present the proof of the Unreachable Branch Law (L6).

Example 3 (Proof of Unreachable Branch (L6)).

$$\begin{aligned}
& (P \triangleleft b \triangleright (Q \triangleleft b \triangleright R)) && [L2] \\
= & ((Q \triangleleft b \triangleright R) \triangleleft \neg b \triangleright P) && [L3] \\
= & (Q \triangleleft b \wedge \neg b \triangleright (R \triangleleft \neg b \triangleright P)) && [\text{propositional calculus}] \\
= & (Q \triangleleft \text{false} \triangleright (R \triangleleft \neg b \triangleright P)) && [L5] \\
= & (R \triangleleft \neg b \triangleright P) && [L2] \\
= & (P \triangleleft b \triangleright R) && \square
\end{aligned}$$

Implication is, of course, still the basis for reasoning about the correctness of conditionals. We can, however, prove refinement laws that support a compositional reasoning technique.

Law 1 (Refinement to Conditional)

$$P \sqsubseteq (Q \triangleleft b \triangleright R) = (P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg b \wedge R) \quad \square$$

This result allows us to prove the correctness of a conditional by a case analysis on the correctness of each branch. Its proof is as follows.

Proof of Law 1.

$$\begin{aligned}
& P \sqsubseteq (Q \triangleleft b \triangleright R) && [\text{definition of } \sqsubseteq] \\
= & [(Q \triangleleft b \triangleright R) \Rightarrow P] && [\text{definition of conditional}] \\
= & [b \wedge Q \vee \neg b \wedge R \Rightarrow P] && [\text{propositional calculus}] \\
= & [b \wedge Q \Rightarrow P] \wedge [\neg b \wedge R \Rightarrow P] && [\text{definition of } \sqsubseteq, \text{ twice}] \\
= & (P \sqsubseteq b \wedge Q) \wedge (P \sqsubseteq \neg b \wedge R) && \square
\end{aligned}$$

A compositional argument is also available for conjunctions.

Law 2 (Separation of Requirements)

$$((P \wedge Q) \sqsubseteq R) = (P \sqsubseteq R) \wedge (Q \sqsubseteq R) \quad \square$$

We can prove that an implementation satisfies a conjunction of requirements by considering each conjunct separately. The omitted proof is left as an exercise for the interested reader.

Sequence is modelled as relational composition. Two relations may be composed, providing that the output alphabet of the first is the same as the input alphabet of the second, except only for the use of dashes.

$$\begin{aligned} P(v') ; Q(v) &\hat{=} \exists v_0 \bullet P(v_0) \wedge Q(v) & \text{if } \text{out}\alpha P = \text{in}\alpha Q' = \{v'\} \\ \text{in}\alpha(P(v') ; Q(v)) &\hat{=} \text{in}\alpha P \\ \text{out}\alpha(P(v') ; Q(v)) &\hat{=} \text{out}\alpha Q \end{aligned}$$

Composition is associative and distributes backwards through the conditional.

$$\begin{aligned} L1 \quad P ; (Q ; R) &= (P ; Q) ; R & \text{associativity} \\ L2 \quad (P \triangleleft b \triangleright Q) ; R &= ((P ; R) \triangleleft b \triangleright (Q ; R)) & \text{left distribution} \end{aligned}$$

The simple proofs of these laws, and those of a few others in the sequel, are omitted for the sake of conciseness.

The definition of assignment is basically equality; we need, however, to be careful about the alphabet. If $A = \{x, y, \dots, z\}$ and $\alpha e \subseteq A$, where αe is the set of free variables of the expression e , the assignment $x :=_A e$ of expression e to variable x changes only x 's value.

$$\begin{aligned} x :=_A e &\hat{=} (x' = e \wedge y' = y \wedge \dots \wedge z' = z) \\ \alpha(x :=_A e) &\hat{=} A \cup A' \end{aligned}$$

There is a degenerate form of assignment that changes no variable: it's called "skip", and has the following definition.

$$\begin{aligned} \mathbf{I}_A &\hat{=} (v' = v) & \text{if } A = \{v\} \\ \alpha \mathbf{I}_A &\hat{=} A \cup A' \end{aligned}$$

Skip is the identity of sequence.

$$L5 \quad P ; \mathbf{I}_{\alpha P} = P = \mathbf{I}_{\alpha P} ; P \quad \text{unit}$$

We keep the numbers of the laws presented in [6] that we reproduce here.

In theories of programming, nondeterminism may arise in one of two ways: either as the result of run-time factors, such as distributed processing; or as the under-specification of implementation choices. Either way, nondeterminism is modelled by choice; the semantics is simply disjunction.

$$\begin{aligned} P \sqcap Q &\hat{=} P \vee Q & \text{if } \alpha P = \alpha Q \\ \alpha(P \sqcap Q) &\hat{=} \alpha P \end{aligned}$$

The alphabet must be the same for both arguments.

The following law gives an important property of refinement: if P is refined by Q , then offering the choice between P and Q is immaterial; conversely, if the choice between P and Q behaves exactly like P , so that the extra possibility of choosing Q does not add any extra behaviour, then Q is a refinement of P .

Law 3 (Refinement and Nondeterminism)

$$P \sqsubseteq Q = (P \sqcap Q = P) \quad \square$$

Proof.

$$\begin{aligned}
 P \sqcap Q &= P && \text{[antisymmetry]} \\
 &= (P \sqcap Q \sqsubseteq P) \wedge (P \sqsubseteq P \sqcap Q) && \text{[definition of } \sqsubseteq, \text{ twice]} \\
 &= [P \Rightarrow P \sqcap Q] \wedge [P \sqcap Q \Rightarrow P] && \text{[definition of } \sqcap, \text{ twice]} \\
 &= [P \Rightarrow P \vee Q] \wedge [P \vee Q \Rightarrow P] && \text{[propositional calculus]} \\
 &= \text{true} \wedge [P \vee Q \Rightarrow P] && \text{[propositional calculus]} \\
 &= [Q \Rightarrow P] && \text{[definition of } \sqsubseteq] \\
 &= P \sqsubseteq Q && \square
 \end{aligned}$$

Another fundamental result is that reducing nondeterminism leads to refinement.

Law 4 (Thin Nondeterminism)

$$P \sqcap Q \sqsubseteq P \quad \square$$

The proof is immediate from properties of the propositional calculus.

Variable blocks are split into the commands $\text{var } x$, which declares and introduces x in scope, and $\text{end } x$, which removes x from scope. Their definitions are presented below, where A is an alphabet containing x and x' .

$$\begin{aligned}
 \text{var } x &\triangleq (\exists x \bullet \mathbf{I}_A) & \alpha(\text{var } x) &\triangleq A \setminus \{x\} \\
 \text{end } x &\triangleq (\exists x' \bullet \mathbf{I}_A) & \alpha(\text{end } x) &\triangleq A \setminus \{x'\}
 \end{aligned}$$

The relation $\text{var } x$ is not homogeneous, since it does not include x in its alphabet, but it does include x' ; similarly, $\text{end } x$ includes x , but not x' .

The results below state that following a variable declaration by a program Q makes x local in Q ; similarly, preceding a variable undeclaration by a program Q makes x' local.

$$\begin{aligned}
 (\text{var } x ; Q) &= (\exists x \bullet Q) \\
 (Q ; \text{end } x) &= (\exists x' \bullet Q)
 \end{aligned}$$

More interestingly, we can use $\text{var } x$ and $\text{end } x$ to specify a variable block.

$$(\text{var } x ; Q ; \text{end } x) = (\exists x, x' \bullet Q)$$

In programs, we use $\text{var } x$ and $\text{end } x$ paired in this way, but the separation is useful for reasoning.

The following laws are representative.

$$L6 \quad (\text{var } x ; \text{end } x) = \mathbf{I}$$

$$L8 \quad (x := e ; \text{end } x) = (\text{end } x)$$

Variable blocks introduce the possibility of writing programs and equations like that below.

$$\begin{aligned} & (\text{var } x ; x := 2 * y ; w := 0 ; \text{end } x) \\ & = (\text{var } x ; x := 2 * y ; \text{end } x) ; w := 0 \end{aligned}$$

Clearly, the assignment to w may be moved out of the scope of the the declaration of x , but what is the alphabet in each of the assignments to w ? If the only variables are w , x , and y , and suppose that $A = \{w, y, w', y'\}$, then the assignment on the right has the alphabet A ; but the alphabet of the assignment on the left must also contain x and x' , since they are in scope. There is an explicit operator for making alphabet modifications such as this: *alphabet extension*. If the right-hand assignment is $P \hat{=} w :=_A 0$, then the left-hand assignment is denoted by P_{+x} .

$$\begin{aligned} P_{+x} & \hat{=} P \wedge x' = x && \text{for } x, x' \notin \alpha P \\ \alpha(P_{+x}) & \hat{=} \alpha P \cup \{x, x'\} \end{aligned}$$

If Q does not mention x , then the following laws hold.

$$L1 \quad \text{var } x ; Q_{+x} ; P ; \text{end } x = Q ; \text{var } x ; P ; \text{end } x$$

$$L2 \quad \text{var } x ; P ; Q_{+x} ; \text{end } x = \text{var } x ; P ; \text{end } x ; Q$$

Together with the laws for variable declaration and undeclaration, the laws of alphabet extension allow for program transformations that introduce new variables and assignments to them.

3 The Complete Lattice

The refinement ordering is a partial order: reflexive, anti-symmetric, and transitive. Moreover, the set of alphabetised predicates with a particular alphabet A is a complete lattice under the refinement ordering. Its bottom element is denoted $\perp_A$, and is the weakest predicate *true*; this is the program that aborts, and behaves quite arbitrarily. The top element is denoted $\top^A$, and is the strongest predicate *false*; this is the program that performs miracles and implements every specification. These properties of abort and miracle are captured in the following two laws, which hold for all P with alphabet A .

$$L1 \quad \perp_A \sqsubseteq P \quad \text{bottom element}$$

$$L2 \quad P \sqsubseteq \top_A \quad \text{top element}$$

The least upper bound is not defined in terms of the relational model, but by the law *L1* below. This law alone is enough to prove laws *L1A* and *L1B*, which are actually more useful in proofs.

- L1 $P \sqsubseteq (\sqcap S)$ iff $(P \sqsubseteq X \text{ for all } X \text{ in } S)$ *unbounded nondeterminism*
 L1A $(\sqcap S) \sqsubseteq X$ for all X in S *lower bound*
 L1B if $P \sqsubseteq X$ for all X in S , then $P \sqsubseteq (\sqcap S)$ *greatest lower bound*

These laws characterise basic properties of least upper bounds.

A function F is *monotonic* if and only if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$. Operators like conditional and sequence are monotonic; negation and conjunction are not. There is a class of operators that are all monotonic.

Example 4 (Disjunctivity and monotonicity). Suppose that $P \sqsubseteq Q$ and that $\odot$ is disjunctive, or rather, $R \odot (S \sqcap T) = (R \odot S) \sqcap (R \odot T)$. From this, we can conclude that $P \odot R$ is monotonic in its first argument.

$$\begin{aligned}
 & P \odot R && [\text{assumption } (P \sqsubseteq Q) \text{ and Law 3}] \\
 = & (P \sqcap Q) \odot R && [\text{assumption } (\odot \text{ disjunctive})] \\
 = & (P \odot R) \sqcap (Q \odot R) && [\text{thin nondeterminism}] \\
 \sqsubseteq & Q \odot R
 \end{aligned}$$

A symmetric argument shows that $P \odot Q$ is also monotonic in its other argument. In summary, disjunctive operators are always monotonic. The converse is not true: monotonic operators are not always disjunctive. $\square$

Since alphabetised relations form a complete lattice, every construction defined solely using monotonic operators has a fixed-point. Even more, a result by Tarski says that the set of fixed-points form a complete lattice themselves. The extreme points in this lattice are often of interest; for example, $\top$ is the strongest fixed-point of $X = P ; X$, and $\perp$ is the weakest.

The weakest fixed-point of the function F is denoted by μF , and is simply the greatest lower bound (the *weakest*) of all the fixed-points of F .

$$\mu F \hat{=} \sqcap \{ X \mid F(X) \sqsubseteq X \}$$

The strongest fixed-point νF is the dual of the weakest fixed-point.

Hoare & He use weakest fixed-points to define recursion. They write a recursive program as $\mu X \bullet \mathcal{C}(X)$, where $\mathcal{C}(X)$ is a predicate that is constructed using monotonic operators and the variable X . As opposed to the variables in the alphabet, X stands for a predicate itself, and we call it the recursive variable. Intuitively, occurrences of X in $\mathcal{C}$ stand for recursive calls to $\mathcal{C}$ itself. The definition of recursion is as follows.

$$\mu X \bullet \mathcal{C}(X) \hat{=} \mu F \quad \text{where } F \hat{=} \lambda X \bullet \mathcal{C}(X)$$

The standard laws that characterise weakest fixed-points are valid.

- L1 $\mu F \sqsubseteq Y$ if $F(Y) \sqsubseteq Y$ *weakest fixed-point*
 L2 $[F(\mu F) = \mu F]$ *fixed-point*

L1 establishes that μF is weaker than any fixed-point; L2 states that μF is itself a fixed-point. From a programming point of view, L2 is just the copy rule.

Proof of L1.

$$\begin{aligned}
 & F(Y) \sqsubseteq Y && [\text{set comprehension}] \\
 = & Y \in \{ X \mid F(X) \sqsubseteq X \} && [\text{lattice law L1A}] \\
 \Rightarrow & \sqcap \{ X \mid F(X) \sqsubseteq X \} \sqsubseteq Y && [\text{definition of } \mu F] \\
 = & \mu F \sqsubseteq Y && \square
 \end{aligned}$$

Proof of L2.

$$\begin{aligned}
 & \mu F = F(\mu F) && [\text{mutual refinement}] \\
 = & \mu F \sqsubseteq F(\mu F) \wedge F(\mu F) \sqsubseteq \mu F && [\text{fixed-point law L1}] \\
 \Leftarrow & F(F(\mu F)) \sqsubseteq F(\mu F) \wedge F(\mu F) \sqsubseteq \mu F && [F \text{ monotonic}] \\
 \Leftarrow & F(\mu F) \sqsubseteq \mu F && [\text{definition}] \\
 = & F(\mu F) \sqsubseteq \sqcap \{ X \mid F(X) \sqsubseteq X \} && [\text{lattice law L1B}] \\
 \Leftarrow & \forall X \in \{ X \mid F(X) \sqsubseteq X \} \bullet F(\mu F) \sqsubseteq X && [\text{comprehension}] \\
 = & \forall X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu F) \sqsubseteq X && [\text{transitivity of } \sqsubseteq] \\
 \Leftarrow & \forall X \bullet F(X) \sqsubseteq X \Rightarrow F(\mu F) \sqsubseteq F(X) && [F \text{ monotonic}] \\
 \Leftarrow & \forall X \bullet F(X) \sqsubseteq X \Rightarrow \mu F \sqsubseteq X && [\text{fixed-point law L1}] \\
 = & \text{true} && \square
 \end{aligned}$$

The while loop is written $b * P$: while b is true, execute the program P . This can be defined in terms of the weakest fixed-point of a conditional expression.

$$b * P \triangleq \mu X \bullet ((P ; X) \triangleleft b \triangleright \mathbf{I})$$

Example 5 (Non-termination). If b always remains true, then obviously the loop $b * P$ never terminates, but what is the semantics for this non-termination? The simplest example of such an iteration is $\text{true} * \mathbf{I}$, which has the semantics $\mu X \bullet X$.

$$\begin{aligned}
 & \mu X \bullet X && [\text{definition of least fixed-point}] \\
 = & \sqcap \{ Y \mid (\lambda X \bullet X)(Y) \sqsubseteq Y \} && [\text{function application}] \\
 = & \sqcap \{ Y \mid Y \sqsubseteq Y \} && [\text{relexivity of } \sqsubseteq] \\
 = & \sqcap \{ Y \mid \text{true} \} && [\text{property of } \sqcap] \\
 = & \perp && \square
 \end{aligned}$$

A surprising, but simple, consequence of Example 5 is that a program can recover from a non-terminating loop!

Example 6 (Aborting loop). Suppose that the sole state variable is x and that c is a constant.

$$\begin{aligned}
& (b * P); x := c && \text{[Example 5]} \\
= & \perp; x := c && \text{[definition of } \perp \text{]} \\
= & \text{true}; x := c && \text{[definition of assignment]} \\
= & \text{true}; x' = c && \text{[definition of composition]} \\
= & \exists x_0 \bullet \text{true} \wedge x' = c && \text{[predicate calculus]} \\
= & x' = c && \text{[definition of assignment]} \\
= & x := c && \square
\end{aligned}$$

Example 6 is rather disconcerting: in ordinary programming, there is no recovery from a non-terminating loop. It is the purpose of *designs* to overcome this deficiency in the programming model; we return to this in Section 5.

4 Theories of Program Correctness

In this section, we apply the theory of alphabetised relations to two key ideas in imperative programming: Hoare logic and the weakest precondition calculus.

4.1 Hoare Logic

Hoare logic provides a way to decompose the correctness argument for a program. The *Hoare triple* $p \{Q\} r$ asserts the correctness of program Q against the specification with precondition p and postcondition r :

$$p \{Q\} r \triangleq (p \Rightarrow r') \sqsubseteq Q$$

The logical rules for Hoare logic are very famous. We reproduce some below.

$$\begin{aligned}
L1 & \text{ if } p \{Q\} r \text{ and } p \{Q\} s \text{ then } p \{Q\} (r \wedge s) \\
L3 & \text{ if } p \{Q\} r \text{ then } (p \wedge q) \{Q\} (r \vee s) \\
L4 & r(e) \{x := e\} r(x) && \text{assignment} \\
L6 & \text{ if } p \{Q_1\} s \text{ and } s \{Q_2\} r \text{ then } p \{Q_1 ; Q_2\} r && \text{sequence} \\
L8 & \text{ if } (b \wedge c) \{Q\} c \text{ then } c \{ \nu X \bullet (Q ; X) \triangleleft b \triangleright \mathbf{I} \} (\neg b \wedge c) && \text{iteration} \\
L9 & \text{ false } \{Q\} r \text{ and } p \{Q\} \text{true and } p \{\text{false}\} \text{false and } p \{\mathbf{I}\} p
\end{aligned}$$

The proof rule for iteration uses *strongest fixed-points*. The implications of this are explained below. First, we present a proof for the rule.

Proof of L8. Suppose that $(b \wedge c) \{Q\} c$, and let Y be the overall specification, so that $Y \triangleq c \Rightarrow \neg b' \wedge c'$.

$$\begin{aligned}
& c \{ \nu X \bullet (Q ; X) \triangleleft b \triangleright \mathbf{I} \} (\neg b \wedge c) && \text{[definition of Hoare triple]} \\
= & Y \sqsubseteq \nu X \bullet (Q ; X) \triangleleft b \triangleright \mathbf{I} && \text{[strongest fixed-point L1]}
\end{aligned}$$

$$\begin{aligned}
&\Leftarrow Y \sqsubseteq (Q; Y) \triangleleft b \triangleright \mathbf{I} && [\text{refinement to conditional (Example 1)}] \\
&= (Y \sqsubseteq (b \wedge Q); Y) \wedge (Y \sqsubseteq \neg b \wedge \mathbf{I}) && [\text{definition of } \sqsubseteq] \\
&= (Y \sqsubseteq (b \wedge Q); Y) \wedge [\neg b \wedge \mathbf{I} \Rightarrow (c \Rightarrow \neg b' \wedge c')] \\
&&& [\text{propositional calculus \& definition of } \mathbf{I}] \\
&= (Y \sqsubseteq (b \wedge Q); Y) \wedge \text{true} && [\text{definition of Hoare triple}] \\
&= c \{ b \wedge Q; Y \} (\neg b \wedge c) && [\text{sequential composition (Hoare L6)}] \\
&\Leftarrow (c \{ b \wedge Q \} c) \wedge (c \{ c \Rightarrow \neg b' \wedge c' \} \neg b \wedge c) \\
&&& [\text{assumption and predicate calculus}] \\
&= \text{true} \quad \square
\end{aligned}$$

This simple proof is the advantage in defining the semantics of a loop using the strongest fixed-point. The next example shows its disadvantage.

Example 7 (Non-termination and Hoare logic).

$$\begin{aligned}
&p \{ \text{true} * \mathbf{I} \} q && [\text{strongest fixed-point semantics}] \\
&= p \{ \nu X \bullet (\mathbf{I}; X) \triangleleft \text{true} \triangleright \mathbf{I} \} q && [\text{strongest fixed-point}] \\
&= p \{ \top \} q && [\text{definition of Hoare triple}] \\
&= ((p \Rightarrow q') \sqsubseteq \top) && [\text{top element}] \\
&= \text{true} \quad \square
\end{aligned}$$

This shows that a non-terminating loop is identified with miracle, and so implements any specification. This drawback is the motivation for choosing weakest fixed-points as the semantics of recursion. We have already seen, however, that this also leads to problems.

An example on the use of Hoare logic is presented below.

Example 8 (Hoare logic proof for Swap). Consider the little program that swaps two numbers, using a temporary register.

$$\text{Swap} \hat{=} t := a; a := b; b := t$$

A simple specification for *Swap* names the initial values of a and b , and then requires that they be swapped. The correctness assertion is therefore given by the Hoare triple below.

$$a = A \wedge b = B \{ t := a; a := b; b := t \} a = B \wedge b = A$$

This assertion can be discharged using the rules of Hoare logic. First, we apply the rule for sequence *L6* to decompose the problem into two parts corresponding to the two sub-programs $t := a; a := b$ and $b := t$. This involves inventing an assertion for the state that exists between these two programs. Our choice is $a = B \wedge t = A$ to reflect the fact that a now has the value of b , and t holds the original value of a .

$$a = A \wedge b = B \{ t := a; a := b; b := t \} a = B \wedge b = A$$

$$\Leftarrow \left(\begin{array}{l} a = A \wedge b = B \{t := a; a := b\} a = B \wedge t = A \quad (i) \\ \wedge \\ a = B \wedge t = A \{b := t\} a = B \wedge b = A \quad (ii) \end{array} \right)$$

Now we use *L6* again; this time to decompose the first sub-program (i).

$$(i) \Leftarrow \left(\begin{array}{l} a = A \wedge b = B \{t := a\} b = B \wedge t = A \quad (iii) \\ \wedge \\ b = B \wedge t = A \{a := b\} a = B \wedge t = A \quad (iv) \end{array} \right)$$

Each of the remaining assertions (ii–iv) is discharged by an application of the rule for assignment, *L4*. $\square$

This example shows how the correctness argument is structured by the application of each rule. Another way of using the rules is to assert only the postcondition; the precondition may then be *calculated* using the rules. We address this topic below.

4.2 Weakest Precondition Calculus

If we fix the program and the postcondition, then we can calculate an appropriate precondition to form a valid Hoare triple. As there will typically be many such preconditions, it is useful to find just one that can lead us to the others. From Hoare Logic Law *L3*, we have that if $p \{Q\} r$, then $(p \wedge q) \{Q\} r$. If we find the *weakest precondition* w that satisfies the Hoare triple $w \{Q\} r$, then this law states that every stronger precondition must also satisfy the assertion.

To find w , we must manipulate the assertion to constrain the precondition to be at least as strong as some other condition. We parametrise p , Q , and r to make their alphabets explicit. The derivation expands the definition of the triple and of refinement, so that the precondition $p(v)$ can be pushed into the antecedent of an implication. The rest of the derivation is simply tidying up.

$$\begin{aligned} & p(v) \{ Q(v, v') \} r(v) && \text{[definition of Hoare triple]} \\ = & (p(v) \Rightarrow r(v')) \sqsubseteq Q(v, v') && \text{[definition of } \sqsubseteq \text{]} \\ = & [Q(v, v') \Rightarrow (p(v) \Rightarrow r(v'))] && \text{[trading antecedents]} \\ = & [p(v) \Rightarrow (Q(v, v') \Rightarrow r(v'))] && \text{[restricting the quantification of } v' \text{]} \\ = & [p(v) \Rightarrow (\forall v' \bullet Q(v, v') \Rightarrow r')] && \text{[De Morgan's quantifier rule]} \\ = & [p(v) \Rightarrow \neg (\exists v' \bullet Q(v, v') \wedge \neg r(v'))] && \text{[change of variable]} \\ = & [p(v) \Rightarrow \neg (\exists v_0 \bullet Q(v, v_0) \wedge \neg r(v_0))] && \text{[definition of sequence]} \\ = & [p(v) \Rightarrow \neg (Q(v, v') ; \neg r(v))] \end{aligned}$$

This says that if p holds, then it is impossible for Q to arrive in a state where r fails to hold. Every precondition must have this property; including, of course, $\neg (Q ; \neg r)$ itself. We can summarise this derivation as follows.

$$\text{if } w = \neg (Q ; \neg r) \text{ then } w \{ Q \} r$$

The condition w is the *weakest solution* for the precondition for program Q to be guaranteed to achieve postcondition r . This useful result motivates and justifies the definition of weakest precondition.

$$Q \text{ wp } r \triangleq \neg (Q ; \neg r)$$

The laws below state the standard weakest precondition semantics for the programming operators.

L1	$((x := e) \text{ wp } r(x)) = r(e)$	<i>assignment</i>
L2	$((P ; Q) \text{ wp } r) = (P \text{ wp } (Q \text{ wp } r))$	<i>sequential composition</i>
L3	$((P \triangleleft b \triangleright Q) \text{ wp } r) = ((P \text{ wp } r) \triangleleft b \triangleright (Q \text{ wp } r))$	<i>conditional</i>
L4	$((P \sqcap Q) \text{ wp } r) = (P \text{ wp } r) \wedge (Q \text{ wp } r)$	<i>nondeterministic choice</i>

Weakest precondition and Hoare logic, however, do not solve the pending issue of non-termination, to which we turn our attention now.

5 Designs

The problem pointed out in Section 2 can be explained as the failure of general alphabetised predicates P to satisfy the equation below.

$$\text{true} ; P = \text{true}$$

In particular, in Example 6 we presented a non-terminating loop which, when followed by an assignment, behaves like the assignment. Operationally, it is as though the non-terminating loop could be ignored.

The solution is to consider a subset of the alphabetised predicates in which a particular observational variable, called *okay*, is used to record information about the start and termination of programs. The above equation holds for predicates P in this set. As an aside, we observe that *false* cannot possibly belong to this set, since $\text{false} = \text{false} ; \text{true}$.

The predicates in this set are called designs. They can be split into precondition-postcondition pairs, and are in the same spirit as specification statements used in refinement calculi. As such, they are a basis for unifying languages and methods like B [1], VDM [7], Z, and refinement calculi [8,2,9].

In designs, *okay* records that the program has started, and *okay'* records that it has terminated. These are auxiliary variables, in the sense that they appear in a design's alphabet, but they never appear in code or in preconditions and postconditions.

In implementing a design, we are allowed to assume that the precondition holds, but we have to fulfill the postcondition. In addition, we can rely on the program being started, but we must ensure that the program terminates. If the precondition does not hold, or the program does not start, we are not committed to establish the postcondition nor even to make the program terminate.

A design with precondition P and postcondition Q , for predicates P and Q not containing $okay$ or $okay'$, is written $(P \vdash Q)$. It is defined as follows.

$$(P \vdash Q) \hat{=} (okay \wedge P \Rightarrow okay' \wedge Q)$$

If the program starts in a state satisfying P , then it will terminate, and on termination Q will be true.

Abort and miracle are defined as designs in the following examples. Abort has precondition *false* and is never guaranteed to terminate.

Example 9 (Abort).

$$\begin{aligned}
 & false \vdash false && [definition\ of\ design] \\
 = & okay \wedge false \Rightarrow okay' \wedge false && [false\ zero\ for\ conjunction] \\
 = & false \Rightarrow okay' \wedge false && [vacuous\ implication] \\
 = & true && [vacuous\ implication] \\
 = & false \Rightarrow okay' \wedge true && [false\ zero\ for\ conjunction] \\
 = & okay \wedge false \Rightarrow okay' \wedge true && [definition\ of\ design] \\
 = & false \vdash true && \square
 \end{aligned}$$

Miracle has precondition *true*, and establishes the impossible: *false*.

Example 10 (Miracle).

$$\begin{aligned}
 & true \vdash false && [definition\ of\ design] \\
 = & okay \wedge true \Rightarrow okay' \wedge false && [true\ unit\ for\ conjunction] \\
 = & okay \Rightarrow false && [contradiction] \\
 = & \neg okay && \square
 \end{aligned}$$

A reassuring result about a design is the fact that refinement amounts to either weakening the precondition, or strengthening the postcondition in the presence of the precondition. This is established by the result below.

Law 5 *Refinement of Designs*

$$P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 = [P_1 \wedge Q_2 \Rightarrow Q_1] \wedge [P_1 \Rightarrow P_2] \quad \square$$

Proof.

$$\begin{aligned}
 & P_1 \vdash Q_1 \sqsubseteq P_2 \vdash Q_2 && [definition\ of\ \sqsubseteq] \\
 = & [(P_2 \vdash Q_2) \Rightarrow (P_1 \vdash Q_1)] && [definition\ of\ design,\ twice] \\
 = & [(okay \wedge P_2 \Rightarrow okay' \wedge Q_2) \Rightarrow (okay \wedge P_1 \Rightarrow okay' \wedge Q_1)] && [case\ analysis\ on\ okay] \\
 = & [(P_2 \Rightarrow okay' \wedge Q_2) \Rightarrow (P_1 \Rightarrow okay' \wedge Q_1)] && [case\ analysis\ on\ okay']
 \end{aligned}$$

$$\begin{aligned}
&= [((P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1)) \wedge (\neg P_2 \Rightarrow \neg P_1)] \quad [\text{propositional calculus}] \\
&= [((P_2 \Rightarrow Q_2) \Rightarrow (P_1 \Rightarrow Q_1)) \wedge (P_1 \Rightarrow P_2)] \quad [\text{predicate calculus}] \\
&= [P_1 \wedge Q_2 \Rightarrow Q_1] \wedge [P_1 \Rightarrow P_2] \quad \square
\end{aligned}$$

The most important result, however, is that abort is a zero for sequence. This was, after all, the whole point for the introduction of designs.

$$L1 \quad \text{true} ; (P \vdash Q) = \text{true} \quad \text{left-zero}$$

Proof.

$$\begin{aligned}
&\text{true} ; (P \vdash Q) \quad [\text{property of sequential composition}] \\
&= \exists \text{okay}_0 \bullet \text{true} ; (P \vdash Q)[\text{okay}_0/\text{okay}] \quad [\text{case analysis}] \\
&= (\text{true} ; (P \vdash Q)[\text{true}/\text{okay}]) \vee (\text{true} ; (P \vdash Q)[\text{false}/\text{okay}]) \\
&\quad [\text{property of design}] \\
&= (\text{true} ; (P \vdash Q)[\text{true}/\text{okay}]) \vee (\text{true} ; \text{true}) \quad [\text{relational calculus}] \\
&= (\text{true} ; (P \vdash Q)[\text{true}/\text{okay}]) \vee \text{true} \quad [\text{propositional calculus}] \\
&= \text{true} \quad \square
\end{aligned}$$

In this new setting, it is necessary to redefine assignment and skip, as those introduced previously are not designs.

$$\begin{aligned}
(x := e) &\hat{=} (\text{true} \vdash x' = e \wedge y' = y \wedge \dots \wedge z' = z) \\
\mathbf{I}_D &\hat{=} (\text{true} \vdash \mathbf{I})
\end{aligned}$$

Their existing laws hold, but it is necessary to prove them again, as their definitions changed.

$$\begin{aligned}
L2 \quad (v := e ; v := f(v)) &= (v := f(e)) \\
L3 \quad (v := e ; (P \triangleleft b(v) \triangleright Q)) &= ((v := e ; P) \triangleleft b(e) \triangleright (v := e ; Q)) \\
L4 \quad (\mathbf{I}_D ; (P \vdash Q)) &= (P \vdash Q)
\end{aligned}$$

As as an example, we present the proof of L2.

Proof of L2.

$$\begin{aligned}
&v := e ; v := f(v) \quad [\text{definition of assignment, twice}] \\
&= (\text{true} \vdash v' = e) ; (\text{true} \vdash v' = f(v)) \quad [\text{case analysis on okay}_0] \\
&= ((\text{true} \vdash v' = e)[\text{true}/\text{okay}'] ; (\text{true} \vdash v' = f(v))[\text{true}/\text{okay}]) \vee \\
&\quad \neg \text{okay} ; \text{true} \quad [\text{definition of design}] \\
&= ((\text{okay} \Rightarrow v' = e) ; (\text{okay}' \wedge v' = f(v))) \vee \neg \text{okay} \quad [\text{relational calculus}] \\
&= \text{okay} \Rightarrow (v' = e ; (\text{okay}' \wedge v' = f(v))) \quad [\text{assignment composition}]
\end{aligned}$$

$$\begin{aligned}
&= okay \Rightarrow okay' \wedge v' = f(e) && \text{[definition of design]} \\
&= (true \vdash v' = f(e)) && \text{[definition of assignment]} \\
&= v := f(e) && \square
\end{aligned}$$

If any of the program operators are applied to designs, then the result is also a design. This follows from the laws below, for choice, conditional, sequence, and recursion. The choice between two designs is guaranteed to terminate when they both are; since either of them may be chosen, then either postcondition may be established.

$$T1 \quad ((P_1 \vdash Q_1) \sqcap (P_2 \vdash Q_2)) = (P_1 \wedge P_2 \vdash Q_1 \vee Q_2)$$

If the choice between two designs depends on a condition b , then so do the precondition and the postcondition of the resulting design.

$$\begin{aligned}
T2 \quad &((P_1 \vdash Q_1) \triangleleft b \triangleright (P_2 \vdash Q_2)) \\
&= ((P_1 \triangleleft b \triangleright P_2) \vdash (Q_1 \triangleleft b \triangleright Q_2))
\end{aligned}$$

A sequence of designs $(P_1 \vdash Q_1)$ and $(P_2 \vdash Q_2)$ terminates when P_1 holds, and Q_1 is guaranteed to establish P_2 . On termination, the sequence establishes the composition of the postconditions.

$$\begin{aligned}
T3 \quad &((P_1 \vdash Q_1); (P_2 \vdash Q_2)) \\
&= ((\neg(\neg P_1; true) \wedge (Q_1 \text{ wp } P_2)) \vdash (Q_1; Q_2))
\end{aligned}$$

Preconditions can be relations, and this fact complicates the statement of Law T3; if the P_1 is a condition instead, then the law is simplified as follows.

$$T3' \quad ((p_1 \vdash Q_1); (P_2 \vdash Q_2)) = (p_1 \wedge (Q_1 \text{ wp } P_2)) \vdash (Q_1; Q_2)$$

A recursively defined design has as its body a function on designs; as such, it can be seen as a function on precondition-postcondition pairs (X, Y) . Moreover, since the result of the function is itself a design, it can be written in terms of a pair of functions F and G , one for the precondition and one for the postcondition.

As the recursive design is executed, the precondition F is required to hold over and over again. The strongest recursive precondition so obtained has to be satisfied, if we are to guarantee that the recursion terminates. Similarly, the postcondition is established over and over again, in the context of the precondition. The weakest result that can possibly be obtained is that which can be guaranteed by the recursion.

$$\begin{aligned}
T4 \quad &(\mu X, Y \bullet (F(X, Y) \vdash G(X, Y))) = (P(Q) \vdash Q) \\
&\text{where } P(Y) = (\nu X \bullet F(X, Y)) \text{ and } Q = (\mu Y \bullet P(Y) \Rightarrow G(P(Y), Y))
\end{aligned}$$

Further intuition comes from the realisation that we want the least refined fixed-point of the pair of functions. That comes from taking the strongest precondition, since the precondition of every refinement must be weaker, and the weakest postcondition, since the postcondition of every refinement must be stronger.

Like the set of general alphabetised predicates, designs form a complete lattice. We have already presented the top and the bottom (miracle and abort).

$$\top_D \hat{=} (\text{true} \vdash \text{false}) = \neg \text{okay}$$

$$\perp_D \hat{=} (\text{false} \vdash \text{true}) = \text{true}$$

The least upper bound and the greatest lower bound are established in the following theorem.

Theorem 1. *Meets and joins*

$$\sqcap_i (P_i \vdash Q_i) = (\bigwedge_i P_i) \vdash (\bigvee_i Q_i)$$

$$\sqcup_i (P_i \vdash Q_i) = (\bigvee_i P_i) \vdash (\bigwedge_i P_i \Rightarrow Q_i)$$

As with the binary choice, the choice $\sqcap_i (P_i \vdash Q_i)$ terminates when all the designs do, and it establishes one of the possible postconditions. The least upper bound models a form of choice that is conditioned by termination: only the terminating designs can be chosen. The choice terminates if any of the designs does, and the postcondition established is that of any of the terminating designs.

6 Healthiness Conditions

Another way of characterising the set of designs is by imposing healthiness conditions on the alphabetised predicates. Hoare & He identify four healthiness conditions that they consider of interest: *H1* to *H4*. We discuss each of them.

6.1 H1: Unpredictability

A relation R is *H1* healthy if and only if $R = (\text{okay} \Rightarrow R)$. This means that observations cannot be made before the program has started. A consequence is that R satisfies the left-zero and unit laws below.

$$\text{true} ; R = \text{true} \quad \text{and} \quad \mathbf{I}_D ; R = R$$

We now present a proof of these results.

Designs with left-units and left-zeros are H1.

$$\begin{aligned}
 R & & [\text{assumption } (\mathbf{I}_D \text{ is left-unit})] \\
 = \mathbf{I}_D ; R & & [\mathbf{I}_D \text{ definition}] \\
 = (\text{true} \vdash \mathbf{I}_D) ; R & & [\text{design definition}] \\
 = (\text{okay} \Rightarrow \text{okay}' \wedge \mathbf{I}) ; R & & [\text{relational calculus}] \\
 = (\neg \text{okay} ; R) \vee (\mathbf{I} ; R) & & [\text{relational calculus}] \\
 = (\neg \text{okay} ; \text{true} ; R) \vee (\mathbf{I} ; R) & & [\text{assumption } (\text{true is left-zero})] \\
 = \neg \text{okay} \vee (\mathbf{I} ; R) & & [\text{assumption } (\mathbf{I} \text{ is left-unit})] \\
 = \neg \text{okay} \vee R & & [\text{relational calculus}] \\
 = \text{okay} \Rightarrow R & & \square
 \end{aligned}$$

H1 designs have a left-zero.

$$\begin{aligned}
 & \text{true} ; R && [\text{assumption } (R \text{ is } H1)] \\
 = & \text{true} ; (okay \Rightarrow R) && [\text{relational calculus}] \\
 = & (\text{true} ; \neg okay) \vee (\text{true} ; R) && [\text{relational calculus}] \\
 = & \text{true} \vee (\text{true} ; R) && [\text{relational calculus}] \\
 = & \text{true} && \square
 \end{aligned}$$

H1 designs have a left-unit.

$$\begin{aligned}
 & \mathbf{I}_D ; R && [\text{definition of } \mathbf{I}_D] \\
 = & (\text{true} \vdash \mathbf{I}_D) ; R && [\text{definition of design}] \\
 = & (okay \Rightarrow okay' \wedge \mathbf{I}) ; R && [\text{relational calculus}] \\
 = & (\neg okay ; R) \vee (okay \wedge R) && [\text{relational calculus}] \\
 = & (\neg okay ; \text{true} ; R) \vee (okay \wedge R) && [\text{true is left-zero}] \\
 = & (\neg okay ; \text{true}) \vee (okay \wedge R) && [\text{relational calculus}] \\
 = & \neg okay \vee (okay \wedge R) && [\text{relational calculus}] \\
 = & okay \Rightarrow R && [R \text{ is } H1] \\
 = & R && \square
 \end{aligned}$$

This means that we could use the left-zero and unit laws to characterise *H1*.

6.2 H2: Possible Termination

The second healthiness condition is $[R[\text{false}/okay'] \Rightarrow R[\text{true}/okay']]$. This means that if R is satisfied when $okay'$ is *false*, it is also satisfied then $okay'$ is *true*. In other words, R cannot *require* nontermination, so that it is always possible to terminate.

The designs are exactly those relations that are *H1* and *H2* healthy. First we present a proof that relations that are *H1* and *H2* healthy are designs.

H1 and H2 healthy relations are designs. Let $R^f = R[\text{false}/okay']$ and $R^t = R[\text{true}/okay']$.

$$\begin{aligned}
 & R && [\text{assumption } (R \text{ is } H1)] \\
 = & okay \Rightarrow R && [\text{propositional calculus}] \\
 = & okay \Rightarrow (\neg okay' \wedge R^f) \vee (okay' \wedge R^t) && [\text{assumption } (R \text{ is } H2)] \\
 = & okay \Rightarrow (\neg okay' \wedge R^f \wedge R^t) \vee (okay' \wedge R^t) && [\text{propositional calculus}] \\
 = & okay \Rightarrow (((\neg okay' \wedge R^f) \vee okay') \wedge R^t) && [\text{propositional calculus}] \\
 = & okay \Rightarrow ((R^f \vee okay') \wedge R^t) && [\text{propositional calculus}] \\
 = & okay \Rightarrow (R^f \wedge R^t) \vee (okay' \wedge R^t) && [\text{assumption } (R \text{ is } H2)]
 \end{aligned}$$

$$\begin{aligned}
&= okay \Rightarrow R^f \vee (okay' \wedge R^t) && [propositional\ calculus] \\
&= okay \wedge \neg R^f \Rightarrow okay' \wedge R^t && [design\ definition] \\
&= \neg R^f \vdash R^t && \square
\end{aligned}$$

It is very simple to prove that designs are *H1* healthy; we present the proof that designs are *H2* healthy.

Designs are H2.

$$\begin{aligned}
&(P \vdash Q)[false/okay'] && [definition\ of\ design] \\
&= (okay \wedge P \Rightarrow false) && [propositional\ calculus] \\
&\Rightarrow (okay \wedge P \Rightarrow Q) && [definition\ of\ design] \\
&= (P \vdash Q)[true/okay'] && \square
\end{aligned}$$

While *H1* characterises the rôle of *okay*, *H2* characterises *okay'*. Therefore, it should not be a surprise that, together, they identify the designs.

6.3 H3: Dischargeable Assumptions

The healthiness condition *H3* is specified as an algebraic law: $R = R ; \mathbf{I}_D$. A design satisfies *H3* exactly when its precondition is a condition. This is a very desirable property, since restrictions imposed on dashed variables in a precondition can never be discharged by previous or successive components. For example, $x' = 2 \vdash true$ is a design that can either terminate and give an arbitrary value to x , or it can give the value 2 to x , in which case it is not required to terminate. This is a rather bizarre behaviour.

A design is H3 iff its assumption is a condition.

$$\begin{aligned}
&((P \vdash Q) = ((P \vdash Q) ; \mathbf{I}_D)) && [definition\ of\ design-skip] \\
&= ((P \vdash Q) = ((P \vdash Q) ; (true \vdash \mathbf{I}_D))) && [sequence\ of\ designs] \\
&= ((P \vdash Q) = (\neg(\neg P ; true) \wedge \neg(Q ; \neg true) \vdash Q ; \mathbf{I}_D)) && [skip\ unit] \\
&= ((P \vdash Q) = (\neg(\neg P ; true) \vdash Q)) && [design\ equality] \\
&= (\neg P = \neg P ; true) && [propositional\ calculus] \\
&= (P = P ; true) && \square
\end{aligned}$$

The final line of this proof states that $P = \exists v' \bullet P$, where v' is the output alphabet of P . Thus, none of the after-variables' values are relevant: P is a condition only on the before-variables.

6.4 H4: Feasibility

The final healthiness condition is also algebraic: $R ; true = true$. Using the definition of sequence, we can establish that this is equivalent to $\exists v' \bullet R$, where

v' is the output alphabet of R . In words, this means that for *every* initial value of the observational variables on the input alphabet, there exist final values for the variables of the output alphabet: more concisely, establishing a final state is feasible. The design $\top_D$ is not *H4* healthy, since miracles are not feasible.

7 Theories of Program Correctness Revisited

In this section, we reconsider our theories of program correctness in the light of the theory of designs. We start with assertional reasoning, which we postponed until we had an adequate treatment of termination. We review Hoare logic and weakest preconditions, before introducing the refinement calculus.

7.1 Assertional Reasoning

A well-established reasoning technique for correctness is that of assertional reasoning. It uses assumptions and assertions to annotate programs: write conditions that must, or are expected to, hold in several points of the program. If the conditions do hold, assumptions and assertions do not affect the behaviour of the program; they are comments. If the condition of an assumption does not hold, the program becomes miraculous; if the condition of an assertion does not hold, the program aborts.

$$\begin{aligned} c^\top &\triangleq \mathbf{I}_D \triangleleft c \triangleright \top_D && [\text{assumption}] \\ c_\perp &\triangleq \mathbf{I}_D \triangleleft c \triangleright \perp_D && [\text{assertion}] \end{aligned}$$

For simplicity, we ignore the alphabets in these definitions.

The following law establishes that a sequence of assertions can be joined.

Law 6 (Composition of Assertions)

$$b_\perp ; c_\perp = (b \wedge c)_\perp \quad \square$$

Proof.

$$\begin{aligned} &b_\perp ; c_\perp && [\text{definition of assertion}] \\ &= (\mathbf{I}_D \triangleleft b \triangleright \perp_D) ; c_\perp && [\text{composition left-distribution over conditional}] \\ &= ((\mathbf{I}_D ; c_\perp) \triangleleft b \triangleright (\perp_D ; c_\perp)) && [\text{left-unit for sequence (L4)}] \\ &= (c_\perp \triangleleft b \triangleright (\perp_D ; c_\perp)) && [\perp_D \text{ left-zero for sequence (L1)}] \\ &= (c_\perp \triangleleft b \triangleright \perp_D) && [\text{definition of assertion}] \\ &= ((\mathbf{I}_D \triangleleft c \triangleright \perp_D) \triangleleft b \triangleright \perp_D) && [\text{conditional associativity (L3)}] \\ &= (\mathbf{I}_D \triangleleft b \wedge c \triangleright \perp_D) && [\text{definition of assertion}] \\ &= (b \wedge c)_\perp && \square \end{aligned}$$

Reasoning with assertions often involves distributing them through a program. For example, we can move an assertion over an assignment.

Law 7 (Assertions and Assignments)

$$(c(e)_{\perp} ; x := e) = (x := e ; c(x)_{\perp}) \quad \square$$

Proof.

$$\begin{aligned}
& c(e)_{\perp} ; x := e && [\text{definition of assertion}] \\
= & (\mathbf{I}_D \triangleleft c(e) \triangleright \perp_D) ; x := e && [\text{sequence L2}] \\
= & \mathbf{I}_D ; x := e \triangleleft c(e) \triangleright \perp_D ; x := e && [x := e \text{ is a design (H1)}] \\
= & x := e \triangleleft c(e) \triangleright \perp_D && [x := e \text{ is H3 and H4}] \\
= & x := e ; \mathbf{I}_D \triangleleft c(e) \triangleright x := e ; \perp_D && [\text{design assignment Law L3}] \\
= & x := e ; (\mathbf{I}_D \triangleleft c(x) \triangleright \perp_D) && [\text{definition of assertion}] \\
= & x := e ; c(x)_{\perp} && \square
\end{aligned}$$

Finally, we present below a law for distributing assertions through a conditional.

Law 8 (Assertions and Conditionals)

$$c_{\perp} ; (P \triangleleft b \triangleright Q) = ((b \wedge c)_{\perp} ; P \triangleleft b \triangleright (\neg b \wedge c)_{\perp} ; Q) \quad \square$$

We leave the proof of this law as an exercise.

7.2 Hoare Logic

In Section 4, we define the Hoare triple for relations as follows.

$$p \{Q\} r \hat{=} (p \Rightarrow r') \sqsubseteq Q$$

The next two examples show that this is not appropriate for designs. First, we consider which specifications are satisfied by an aborting program.

Example 11 (Abort).

$$\begin{aligned}
& p \{ \perp_D \}_D r && [\text{definition of } \perp_D] \\
= & p \{ \text{false} \vdash \text{true} \} r && [\text{definition of design}] \\
= & p \{ \text{okay} \wedge \text{false} \Rightarrow \text{okay}' \wedge \text{true} \} r && [\text{propositional calculus}] \\
= & p \{ \text{true} \}_D r && [\text{definition of Hoare triple}] \\
= & [\text{true} \Rightarrow (p \Rightarrow r')] && [\text{propositional calculus}] \\
= & [p \Rightarrow r'] && \square
\end{aligned}$$

This is simply wrong, since it establishes the validity of

$$\text{true} \{ \perp_D \} \text{true}$$

Here the requirement is for the program to terminate in *every* state—which abort clearly does fail to do. Next, we consider which specifications are satisfied by a miraculous program.

Example 12 (Miracle).

$$\begin{aligned}
 & p \{ \top_D \}_D r && \text{[definition of } \top_D \text{]} \\
 = & p \{ \text{true} \vdash \text{false} \} r && \text{[definition of design]} \\
 = & p \{ \text{okay} \wedge \text{true} \Rightarrow \text{okay}' \wedge \text{false} \} r && \text{[propositional calculus]} \\
 = & p \{ \neg \text{okay} \} r && \text{[definition of Hoare triple]} \\
 = & [\neg \text{okay} \Rightarrow (p \Rightarrow r')] && \text{[case analysis]} \\
 = & [\neg \text{true} \Rightarrow (p \Rightarrow r')] \wedge [\neg \text{false} \Rightarrow (p \Rightarrow r')] && \text{[propositional calculus]} \\
 = & [p \Rightarrow r'] && \text{[from Example 11]} \\
 = & p \{ \perp_D \}_D r && \square
 \end{aligned}$$

Again, this is simply wrong, since it is the same result as before—and a miracle is surely different from an aborting program! So, we conclude that we need to adjust the definition of Hoare triple for designs. For any design Q , we define the Hoare triple as follows.

$$p \{ Q \}_D r \triangleq (p \vdash r') \sqsubseteq Q$$

If we replay our two examples, we get the expected results. First, what specifications are satisfied by an aborting program?

Example 13 (Abortive implementation).

$$\begin{aligned}
 & p \{ \perp_D \}_D r && \text{[definition of } \perp_D \text{]} \\
 = & p \{ \text{false} \vdash \text{true} \} r && \text{[definition of Hoare triple]} \\
 = & [(\text{false} \vdash \text{true}) \Rightarrow (p \vdash r')] && \text{[definition of design, twice]} \\
 = & [(p \Rightarrow \text{false}) \wedge (p \wedge \text{true} \Rightarrow r)] && \text{[propositional calculus]} \\
 = & [\neg p \wedge (p \Rightarrow r')] && \text{[propositional calculus]} \\
 = & [\neg p] && \square
 \end{aligned}$$

The answer is that the precondition must be a contradiction. Next, what specifications are satisfied by a miraculous program?

Example 14 (Miraculous implementation).

$$\begin{aligned}
 & p \{ \top_D \}_D r && \text{[definition of } \top_D \text{]} \\
 = & p \{ \text{true} \vdash \text{false} \} r && \text{[definition of Hoare triple]} \\
 = & [(\text{true} \vdash \text{false}) \Rightarrow (p \vdash r')] && \text{[definition of design, twice]} \\
 = & [(p \Rightarrow \text{true}) \wedge (p \wedge \text{false} \Rightarrow r')] && \text{[propositional calculus]} \\
 = & [\text{true}] && \text{[predicate calculus]} \\
 = & \text{true} && \square
 \end{aligned}$$

The answer is that a miracle satisfies *every* specification.

We now prove that Hoare logic rule *L1* holds for the new definition.

$$\begin{aligned}
& (p \{Q\}_D r) \wedge (p \{Q\}_D s) && [\text{definition of design Hoare triple}] \\
= & ((p \vdash r') \sqsubseteq Q) \wedge ((p \vdash s') \sqsubseteq Q) && [\text{separation of requirements}] \\
= & (p \vdash r') \wedge (p \vdash s') \sqsubseteq Q && [\text{definition of designs}] \\
= & (okay \wedge p \Rightarrow okay' \wedge r') \wedge (okay \wedge p \Rightarrow okay' \wedge s') \sqsubseteq Q && [\text{propositional calculus}] \\
= & (okay \wedge p \Rightarrow okay' \wedge okay' \wedge r' \wedge s') \sqsubseteq Q && [\text{definition of design}] \\
= & (p \vdash r' \wedge s') \sqsubseteq Q && [\text{definition of design Hoare triple}] \\
= & p \{Q\}_D r \wedge s && \square
\end{aligned}$$

Other rules may be proved in a similar way.

7.3 Weakest Precondition

Once more, we can use our definition of a Hoare triple to derive an expression for the weakest precondition of *H3* healthy designs.

$$\begin{aligned}
& p \{P \vdash Q\}_D r && [\text{definition of Hoare triple}] \\
= & (p \vdash r') \sqsubseteq (P \vdash Q) && [\text{refinement of designs}] \\
= & [(p \Rightarrow P) \wedge (p \wedge Q \Rightarrow r')] && [\text{propositional calculus}] \\
= & [p \Rightarrow P \wedge (Q \Rightarrow r')] && [\text{predicate calculus}] \\
= & [\forall v' \bullet p \Rightarrow P \wedge (Q \Rightarrow r')] && [\text{since } v' \text{ not free in } a] \\
= & [p \Rightarrow \forall v' \bullet P \wedge (Q \Rightarrow r')] && [\text{assumption } (P \vdash Q \text{ is } H3): v' \text{ not free in } P] \\
= & [p \Rightarrow P \wedge (\forall v' \bullet Q \Rightarrow r')] && [\text{De Morgan's quantifier law}] \\
= & [p \Rightarrow P \wedge \neg (\exists v' \bullet \neg (Q \Rightarrow r'))] && [\text{propositional calculus}] \\
= & [p \Rightarrow P \wedge \neg (\exists v' \bullet Q \wedge \neg r')] && [\text{definition of sequential composition}] \\
= & [p \Rightarrow P \wedge \neg (Q ; \neg r)] && \square
\end{aligned}$$

This motivates our new definition for the weakest precondition for a design.

$$(P \vdash Q) wp_D r = (P \wedge (Q wp r))$$

This new definition uses the *wp* operator introduced before.

7.4 Specification Statements

Our final theory of program correctness is Morgan's refinement calculus [8]. There, a *specification statement* is a kind of design. The syntax is as follows.

$$\text{frame} : [\text{precondition}, \text{postcondition}]$$

The *frame* describes the variables that are allowed to change, and the *precondition* and *postcondition* are the same as those in a design.

For example, the specification statement $x : [y \neq 0, x' = x \text{ div } y]$ is represented by the design $y \neq 0 \vdash (x' = x \text{ div } y)_{+y}$, providing that the only program variables are x and y .

The refinement law for assignment introduction is as shown below.

Law 9 *Assignment Introduction in the Refinement Calculus*

providing that $[p \Rightarrow Q[e/w']]$

$$w, x : [p, Q] \sqsubseteq w := e$$

□

Proof.

$$\begin{aligned}
 w := e & \quad \text{[definition of assignment]} \\
 = \text{true} \vdash (w' = e)_{+\{x,y\}} & \quad \text{[weaken precondition]} \\
 \sqsupseteq p \vdash (w' = e)_{+\{x,y\}} & \quad \text{[assumption]} \\
 = p \vdash (Q[e/w'] \wedge w' = e)_{+\{x,y\}} & \quad \text{[Leibniz]} \\
 = p \vdash (Q \wedge w' = e)_{+\{x,y\}} & \quad \text{[alphabet extension]} \\
 = p \vdash (Q \wedge w' = e \wedge x' = x)_{+y} & \quad \text{[strengthen postcondition]} \\
 \sqsupseteq p \vdash Q_{+y} & \quad \text{[definition of specification statement]} \\
 = w, x : [p, Q] & \quad \square
 \end{aligned}$$

Another important law allows the calculation of conditionals.

Law 10 *Conditional Introduction in the Refinement Calculus*

providing that $[p \Rightarrow \bigvee i \bullet g_i]$

$$w : [p, Q] \sqsubseteq \left\{ \begin{array}{l} \text{if } g_1 \longrightarrow w : [g_1 \wedge p, Q] \\ \square g_2 \longrightarrow w : [g_2 \wedge p, Q] \\ \vdots \\ \square g_n \longrightarrow w : [g_n \wedge p, Q] \\ \text{fi} \end{array} \right.$$

□

This law uses a generalised form of conditional, present in Dijkstra's language of guarded commands [4] and in Morgan's calculus. The conditions are called guards, and the choice of branch to execute is nondeterministic among those whose guards are true. The definition of this guarded conditional is not difficult, but here we consider just the conditional operator we have presented before.

Proof of binary case : $w : [p, Q] \sqsubseteq (w : [g \wedge p, Q] \triangleleft g \triangleright w : [\neg g \wedge p, Q])$.

In order to prove this refinement, we can resort to Law 1 and proceed by case analysis. In this case, we need to prove $w : [p, Q] \sqsubseteq g \wedge w : [g \wedge p, Q]$ and $w : [p, Q] \sqsubseteq \neg g \wedge w : [\neg g \wedge p, Q]$. Below, we prove the first case; the second case is similar.

$$\begin{aligned}
& g \wedge w : [g \wedge p, Q] && \text{[definition of specification statement]} \\
= & g \wedge (g \wedge p \vdash Q_{+x}) && \text{[definition of alphabet extension]} \\
= & g \wedge (g \wedge p \vdash Q \wedge x' = x) && \text{[definition of design]} \\
= & g \wedge (okay \wedge g \wedge p \Rightarrow okay' \wedge Q \wedge x' = x) && \text{[propositional calculus]} \\
= & g \wedge (okay \wedge p \Rightarrow okay' \wedge Q \wedge x' = x) && \text{[propositional calculus]} \\
\Rightarrow & (okay \wedge p \Rightarrow okay' \wedge Q \wedge x' = x) && \text{[definition of design]} \\
= & p \vdash Q_{+x} && \text{[definition of specification statement]} \\
= & w : [p, Q] && \square
\end{aligned}$$

Next, we present an example of the application of the refinement calculus. The problem is to calculate the maximum and minimum of two numbers.

Example 15 (Finding the maximum and the minimum). The problem has a simple specification: $x, y : [true, x' = \max(x, y) \wedge y' = \min(x, y)]$. Our first step is to use Law 10 to introduce a conditional statement that checks the order of the two variables.

$$\begin{aligned}
& x, y : [true, x' = \max(x, y) \wedge y' = \min(x, y)] \\
\sqsubseteq & \left(\begin{array}{l} x, y : [x < y, x' = \max(x, y) \wedge y' = \min(x, y)] \quad (i) \\ \triangleleft x < y \triangleright \\ x, y : [x \geq y, x' = \max(x, y) \wedge y' = \min(x, y)] \quad (ii) \end{array} \right)
\end{aligned}$$

The ‘then’ case is easily implemented using a multiple assignment, a generalised assignment that updates a list of variables in parallel. Its semantics and properties are similar to those of the single assignment; in particular, Law 9 holds.

$$(i) \sqsubseteq x, y := y, x$$

providing that

$$[x < y \Rightarrow (x' = \max(x, y) \wedge y' = \min(x, y))[y, x/x', y']]$$

which follows from substitution and properties of the *max* and *min* functions. The ‘else’ case is even simpler, since the variables are already in the right order.

$$(ii) \sqsubseteq \text{skip}$$

providing that

$$[x \geq y \Rightarrow (x' = \max(x, y) \wedge y' = \min(x, y))[x, y/x', y']]$$

The development may be summarised as the following refinement.

$$\begin{aligned}
 & x, y : [\text{true}, x' = \max(x, y) \wedge y' = \min(x, y)] \\
 \sqsubseteq & \\
 & \text{if } x < y \text{ then } x, y := y, x \text{ else skip fi} \\
 = & \\
 & \text{if } x < y \text{ then } x, y := y, x \text{ fi} \quad \square
 \end{aligned}$$

There are many other laws in the refinement calculus; we omit them for the sake of conciseness.

8 Conclusions

Through a series of examples, we have presented the alphabetised relational calculus and its sub-theory of designs. In this framework, we have presented the formalisation of four different techniques for reasoning about program correctness. The assertional technique, the Hoare logic, and the weakest preconditions are presented in [6]; our original contribution is a recasting of Hoare logic and weakest preconditions in the theory of designs, and an outline of the formalisation of Morgan's calculus.

We hope to have given a didactic and accessible account of this basic foundation of the unifying theories of programming. We have left out, however, most of the more elaborate programming constructs contemplated in [6]. These include theories for concurrency, communication, and functional, logic, and higher-order programming. We also have not discussed their account of algebraic and operational semantics, nor the correctness of compilers.

In our recent work, we have used the theory of communication and concurrency to provide a semantics for *Circus* [13], an integration of Z and CSP [11] aimed at supporting the development of reactive concurrent systems. We have used the semantics to justify a refinement strategy for *Circus* based on calculational laws in the style of Morgan [3].

In [10], UTP is also used to give a semantics to another integration of Z and CSP, which also includes object-oriented features. In [12], UTP is extended with constructs to capture real-time properties as a first step towards a semantic model for a timed version of *Circus*. In [5], a theory of general correctness is characterised as an alternative to designs; instead of *H1* and *H2*, a different healthiness condition is adopted to restrict general relations.

Currently, we are collaborating with colleagues to extend UTP to capture mobility, synchronicity, and object orientation. We hope to contribute to the development of a theory that can support all the major concepts available in modern programming languages.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. R. J. R. Back and J. Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 – 3):146 – 181, 2003.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
5. S. Dunne. Recasting Hoare and He’s Unifying Theories of Programs in the Context of General Correctness. In A. Butterfield and C. Pahl, editors, *IWFM’01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.
6. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
7. C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 1986.
8. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
9. J. M. Morris. A Theoretical Basis for Stepwise Refinement and the Programming Calculus. *Science of Computer Programming*, 9(3):287 – 306, 1987.
10. S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 321 – 340, 2003.
11. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
12. A. Sherif and He Jifeng. Towards a Time Model for *Circus*. In *International Conference in Formal Engineering Methods*, pages 613 – 624, 2002.
13. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.
14. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

An Integration of Program Analysis and Automated Theorem Proving

Bill J. Ellis and Andrew Ireland

School of Mathematical & Computer Sciences
Heriot-Watt University, Edinburgh, Scotland, UK
bill@macs.hw.ac.uk, a.ireland@hw.ac.uk

Abstract. Finding tractable methods for program reasoning remains a major research challenge. Here we address this challenge using an integrated approach to tackle a niche program reasoning application. The application is proving exception freedom, *i. e.* proving that a program is free from run-time exceptions. Exception freedom proofs are a significant task in the development of high integrity software, such as safety and security critical applications. The SPARK approach for the development of high integrity software provides a significant degree of automation in proving exception freedom. However, when the automation fails, user interaction is required. We build upon the SPARK approach to increase the amount of automation available. Our approach involves the integration of two static analysis techniques. We extend the proof planning paradigm with program analysis.

1 Introduction

Program reasoning has been an active area of research since the early days of computer science, as demonstrated by a program proof by Alan Turing [36]. However, as highlighted in [27] the search for “tractable methods” has remained a key research challenge. Here we address this challenge by considering the integration of two distinct static analysis techniques. The first is *proof planning* [4], a theorem proving technique developed by the automated deduction community. The second is *program analysis*, a general technique for automatically discovering interesting properties from a program’s source code.

For our program reasoning we have focused on the SPARK programming language [1]. SPARK is designed for the development of high integrity software, as seen in safety and security critical applications. Our primary interest is in the development of automatic methods for proving exception freedom in SPARK programs, *i.e.* proving that a program is free from run-time exceptions. Such program reasoning represents an important task in the development of high integrity software. For instance, the loss of Ariane 5 was a result of an integer overflow run-time error [15], while buffer overflows are the most common form of security vulnerability [12]. The SPARK toolset supports proof of exception freedom using formal verification. This reduces the task of guaranteeing

exception freedom to proving a number of theorems called *verification conditions* (VCs). Industrial strength evidence [9] shows that the SPARK toolset can typically prove around 90% of such VCs automatically. Our work targets the remaining 10%. These typically account for hundreds of VCs, each requiring user interaction to complete the proof.

Background material on SPARK and the nature of the verification problem being addressed is presented in §2. In §3 we compare proof using the SPARK toolset to proof following our approach. The details of our approach are presented in §4, §5, §6 and §7. In §8 related work is discussed while in §9 progress and future work are outlined. Our conclusions are presented in §10.

2 Background to the Problem

2.1 The SPARK Approach

The SPARK programming language is defined as a subset of Ada [26]. SPARK excludes many Ada constructs, such as pointers, dynamic memory allocation and recursion to make static analysis of SPARK feasible. SPARK includes an annotation language that supports flow analysis and formal proof. In the case of formal proof the annotations capture the program specification, asserting properties that must be true at particular program points. The annotations are supplied within regular Ada comments, allowing a SPARK compliant program to be compiled using any Ada compiler.

Compliance to the SPARK language is enforced by a static analyser called the EXAMINER. In addition, the EXAMINER performs data flow and information flow analysis [3]. The EXAMINER supports formal verification by building directly upon the Floyd/Hoare style of reasoning. VCs can be generated for proofs of both partial correctness and exception freedom. Two additional tools called the SPADE SIMPLIFIER and SPADE PROOF CHECKER are used to prove these VCs. The SIMPLIFIER is a special purpose theorem prover designed to automatically discharge relatively simple VCs while the PROOF CHECKER is an interactive proof development environment.

2.2 SPARK Exception Freedom

By its definition, SPARK eliminates many of the run-time exceptions that can be raised within Ada. However, index, range, division and overflow checks can still raise exceptions in SPARK code. The EXAMINER generates *run-time check* (RTC) VCs to statically guard against such exceptions. The RTC VCs are equivalent to the Ada run-time checks, consequently proving every RTC VC guarantees exception freedom. To generate VCs every loop must be annotated with an invariant. To support proof of exception freedom for sparsely annotated SPARK code, the EXAMINER automatically inserts invariants as will be described in §5.2.

To illustrate the problems associated with proving RTC VCs consider the SPARK code given in Figure 1. Note that this is used as a running example

```

package FilterPackage is
  subtype AR_T is Integer
    range 0..9;
  type A_T is array (AR_T)
    of Integer;
  procedure Filter(A: in A_T;
    R: out Integer);
  --#derives R from A;
end FilterPackage;

```

```

package body FilterPackage is
  procedure Filter(A: in A_T;
    R: out Integer)
  is
  begin
    R:=0;
    for I in AR_T loop
      if A(I)>=0 and A(I)<=100 then
        R:=R+A(I);
      end if;
    end loop;
  end Filter;
end FilterPackage;

```

Note that while `--#` represents an Ada comment it also denotes a SPARK annotation. Here the annotation is describing the flow information that `R` is derived from array `A` in subprogram `Filter`. This specification is checked automatically by the EXAMINER during its information flow analysis.

Fig. 1. Filter and sum values in an array

throughout the paper. Consider the assignment statement in the then-branch, *i.e.* `R:=R+A(I)`, whose corresponding RTC VC is given in Figure 2. There are two aspects to proving that this assignment can not raise an exception. Firstly, we must show that the value of `I` can never exceed the range of array `A`, *i.e.* `C1` and `C2`. Secondly, we must show that the value of the expression `R+A(I)` lies within the legal bounds of `R`, *i.e.* `C3` and `C4`. While proving `C1` and `C2` is trivial (match with `H2` and `H3` respectively), `C3` and `C4` are unprovable. This problem arises as there is insufficient proof context. Note that the RTC VCs involve proving that variables lie inside legal bounds. This is the case for all RTC VCs, allowing us to target our proof techniques and program analysis accordingly.

3 Comparing Proof in SPARK with Our Approach

3.1 Proof via the SPARK Toolset

Completing a program proof in the SPARK toolset typically requires several steps of user interaction. The general proof process undertaken is summarised below.

1. **Incomplete proof:** For each VC yet to be proved the user must determine the reason for the failure, implement a suitable patch, and then repeat this proof process. Three reasons for failure are considered.
 - a) **Insufficient proof context:** The VC is unprovable as the proof context is not sufficiently strong. The user must introduce the required proof context by strengthening the program specification.

```

H1:   for_all (i__1: integer, ((i__1 >= ar_t__first) and
      (i__1 <= ar_t__last)) -> ((element(a, [i__1]) >=
      integer__first) and (element(a, [i__1]) <= integer__last))) .
H2:   loop__1__i >= ar_t__first .
H3:   loop__1__i <= ar_t__last .
H4:   element(a, [loop__1__i]) >= 0 .
H5:   element(a, [loop__1__i]) <= 100 .
      ->
C1:   loop__1__i >= ar_t__first .
C2:   loop__1__i <= ar_t__last .
C3:   r + element(a, [loop__1__i]) >= integer__first .
C4:   r + element(a, [loop__1__i]) <= integer__last .

```

The EXAMINER generates eight VCs for the running example of Figure 1. Three of these are RTC VCs, while the rest are proving properties asserted by the EXAMINER. The RTC VC above corresponds to proving that the assignment in the *then*-branch can never raise an exception. Here H1, H2 and H3 are a result of the invariant automatically inserted by the EXAMINER. Note that *element(a, [i])* denotes accessing array *a* at index *i*.

Fig. 2. A run-time check verification condition (RTC VC)

- b) **Discovered a bug:** The VC can be proved to be false. This indicates the presence of a bug in the source code or specification. The VC will typically give a strong clue as to the nature of the bug. The user must modify the code or specification to eliminate the bug.
 - c) **Beyond the simplifier:** The VC is provable however its proof is beyond the scope of the SIMPLIFIER. The user must prove the VC via an interactive session with the PROOF CHECKER.
2. **Complete proof:** Every VC is discharged by the SIMPLIFIER and any user guided proofs created in the PROOF CHECKER.

This process is rarely intellectually demanding. However, typically many hundreds of proof failures need to be patched per application. Further, all interactive proofs will be tuned to a particular version of a program. As the program is changed these proofs may break and require refinement. Thus this task presents a significant bottle-neck to the practical completion of exception freedom proofs.

3.2 Proof via Our Approach

Our approach reduces the amount of user interaction required to complete a proof. We extend the existing SPARK toolset with a new tool called NUSPADE¹. NUSPADE is a proof planner that also incorporates program analysis. By using NUSPADE aspects of the proof process outlined above can be automated.

¹ The name NUSPADE emphasises that we are building upon SPADE.

1. **Incomplete proof:** Each VC yet to be proved is automatically tackled by NUSPADE. If NUSPADE successfully finds a proof plan then this is exported as a customised tactic for execution inside the PROOF CHECKER. If NUSPADE fails to find a proof plan three situations are possible.
 - a) **Insufficient proof context:** If NUSPADE is able to identify missing proof context then it can exploit the services of a program analysis oracle to enhance the program specification accordingly.
 - b) **Discovered a bug:** If NUSPADE reduces a VC to false then it must indicate a bug. Although not considered further in this paper, there is scope for configuring NUSPADE to actively detect common programming errors. This will involve targeting the forms of VCs that these errors tend to produce with suitable disproving methods.
 - c) **Require user interaction:** If the above cases do not apply, NUSPADE is unable to progress. The user must pursue the interactive proof process outlined above in §3.1.
2. **Complete proof:** Every VC is discharged by the SIMPLIFIER, any tactics created by NUSPADE and any user guided proofs created in the PROOF CHECKER.

4 Proof Planning

Proof planning is an artificial intelligence technique for guiding tactic based theorem provers. It has been extensively investigated within the context of proof by mathematical induction [6]. A proof plan represents the pattern associated with a family of proofs and is used to guide the search for the proof of a given conjecture within the family. A successful search instantiates the proof plan for the given conjecture. From the instantiated proof plan a tactic can be mechanically extracted and automatically checked using an appropriate theorem prover. Adopting this approach passes the burden of soundness to the theorem prover. Free from the constraints of demonstrating soundness, greater flexibility is possible when planning a proof.

A proof plan corresponds to a set of *methods*. Each method expresses preconditions for the applicability of a particular tactic. The methods are typically less expensive to execute and more constrained than their corresponding tactics. Another significant component of proof planning is the *proof critics* mechanism [19,21]. Proof critics are associated with the partial success of proof methods and provide a mechanism for patching failed proofs.

4.1 Exception Freedom Methods

Our exception freedom proof plan contains four methods as outlined below. Note that these appear in the order used within the proof planner, *i.e.* the simpler more immediate methods are tried first. Further, note that the details of these methods will be described in more detail in §7.2.

Preconditions for transitivity method:	Preconditions for transitivity critic:
1. There exists a goal of the form: $E \text{ Rel } C$. 2. For all variables V_i that occur within E there exists a hypothesis of the form: $V_i \text{ Rel } E_i$.	1. Precondition 1 of the transitivity method holds, <i>i.e.</i> there exists a goal of the form: $E \text{ Rel } C$. 2. Precondition 2 of the transitivity method fails, <i>i.e.</i> there exists a variable V_i that occurs within E such that there does not exist a hypothesis of the form: $V_i \text{ Rel } E_i$.

Note that E and C range over expressions and constants respectively, while Rel denotes a transitive relation.

Fig. 3. Preconditions for the transitivity method and critic

1. **elementary**: Applicable to goals that are automatically discharged by the PROOF CHECKER, modulo some minor simplifications.
2. **fertilise**: Applicable where part of a goal matches a hypothesis, producing a simplified goal.
3. **decomposition**: Applicable to a transitive relation within a goal, decomposing its term structure.
4. **transitivity**: Applicable to a goal involving a transitive relation, introducing a transitive step into the proof.

4.2 Exception Freedom Critics

In our exception freedom proof plan a proof critic is associated with the transitivity method. The transitivity critic detects insufficient proof context. It describes the missing proof context using *hypotheses schemata*. The preconditions for the transitivity method and critic are presented in Figure 3. Note that the preconditions of the transitivity critic are expressed in terms of the partial success of the preconditions of the transitivity method.

4.3 Failed Proof Plan

To illustrate the behaviour of our exception freedom proof plan we return to our running example of Figure 1 and its corresponding RTC VC for the *then*-branch shown in Figure 2. We focus on the goal of proving C_4 ,

$$r + \text{element}(a, i) \leq \text{integer\_last} ,$$

noting that the proof context includes the hypothesis H5

$$\text{element}(a, i) \leq 100 .$$

All methods completely fail except the transitivity method which is partially successful. In the following A , B , X and Y range over expressions. The goal satisfies the first precondition of the transitivity method as there exists a goal of the form $E \text{ Rel } C$. However, the second precondition fails, as variable r exists within $r + \text{element}(a, i)$ yet there is no hypothesis matching $r \leq B$. Note that $\text{element}(a, i)$ does not cause the second precondition to fail as there does exist a hypothesis of the form $\text{element}(a, i) \leq Y$. The proof plan for the lower bound of $r + \text{element}(a, i)$ similarly fails as there does not exist a hypothesis that matches $r \geq A$ and there does exist a hypothesis that matches $\text{element}(a, i) \geq X$. Each of these failure patterns trigger the transitivity critic, suggesting the need for additional hypotheses corresponding to the schema

$$(r \geq A) \wedge (r \leq B) .$$

This schema suggests that additional information on the bounds of r needs to be introduced through the discovery of a stronger loop invariant. Below in §5 we describe how this discovery is automated through program analysis.

5 Program Analysis

Program analysis involves automatically calculating interesting properties about source code. Different program analysis techniques have been presented, including flow analysis [3], performance analysis [14] and discovering constraints on variables [11].

Although VCs are generated by combining source code with its specification they typically reveal only a subset of this information. Thus, it is reasonable to return to the source code and its corresponding specification. For example, in [28] invariant discovery is tackled through *top-down* and *bottom-up* approaches, exploiting the specification and source code respectively. Top-down approaches are more applicable in the presence of a strong specification. As exception freedom proofs are typically performed on minimally annotated code the top-down approach is less effective. However, we believe that top-down approaches have a significant role to play in assisting partial correctness proofs [23]. Bottom-up approaches are more applicable where low level implementation detail is desired. This is especially suitable for exception freedom proofs, which involve reasoning about the low level details of an algorithm. Thus we focus on extracting properties from the source code using program analysis.

5.1 Program Analysis Oracle

Program analysis for program verification typically involves the use of heuristic based techniques, as seen in [28,18]. These techniques can be quite unstructured, with different techniques interacting in various ways and often targeting a particular area of a program. In particular, these techniques often produce candidate properties that require nontrivial reasoning in order to prove their correctness.

Thus it is not practical to capture such imprecise techniques in a formal manner. Our strategy is to view program analysis as an oracle. The system produces candidate properties for use during proof planning. The soundness of the entire approach is ensured by the execution of the tactics generated by the proof planner.

We capture distinct program analysis heuristics as *program analysis methods*. Our program analysis begins by first translating the input source code into a flowchart. The program analysis methods are then called in series to annotate the flowchart with *abstract values*, *i. e.* approximate descriptions of program variables. Each method employs a suitable representation to describe these abstract values. Once all of the methods have completed, a collection of *program properties* are extracted from the annotated flowchart. These properties may be accessed during proof planning to assist in the verification effort.

5.2 Program Analysis Performed by the Examiner

As mentioned in §2.2, the EXAMINER automatically inserts invariants to enable proof of exception freedom in minimally annotated SPARK. In addition to this the EXAMINER also inserts preconditions to enrich the program specification. This behaviour captures the spirit of our program analysis, *i. e.* exploiting information in the source code to automatically discover useful properties.

The EXAMINER adds a precondition that every imported subprogram parameter is within its type. Further, the EXAMINER adds a default invariant of true for each loop. This is strengthened by asserting that `for` loop iterators are within their type. Further, any precondition is copied into the invariant adjusting all variables to refer to their initial, rather than current, value.

5.3 Program Analysis Methods

Based on industrial strength examples and focusing on exception freedom proofs a small collection of program analysis methods have been established. These are presented in the following sections. For brevity the examples presented focus on regular program variables. However, they can be naturally extended to deal with arrays and records, the two main SPARK structures.

5.4 Method: Type

SPARK adopts the strong Ada type system, imposing some additional constraints to ease static analysis. As type information directly reveals a variables legal bounds it is especially valuable in exception freedom proofs. For example, consider the source code in Figure 4. The variables `I` and `J` are declared to be of type `ARPO_T`. Thus the method will find abstract values for `I` and `J` which may be expressed using the following candidate invariant. Note that SPARK code assertions, including invariants, are annotated as `--#assert`.

```
--#assert (I>=ARPO_T'First and I<=ARPO_T'Last) and
--#      (J>=ARPO_T'First and J<=ARPO_T'Last);
```

<pre> package PolishFlagPackage is subtype AR_T is Integer range 1..4; type AC_T is (Red, White); type A_T is array (AR_T) of AC_T; procedure PolishFlag(A: in out A_T); --# derives A from A; end PolishFlagPackage; </pre>	<pre> package body PolishFlagPackage is procedure PolishFlag(A: in out A_T) is subtype ARPO_T is Integer range A'First..A'Last+1; I,J: ARPO_T; T: AC_T; begin I:=ARPO_T'First; J:=ARPO_T'Last; loop exit when I=J; if A(I)=Red then I:=I+1; else J:=J-1; T:=A(I); A(I):=A(J); A(J):=T; end if; end loop; end PolishFlag; end PolishFlagPackage; </pre>

Fig.4. Sort two value array

This invariant is required to prove exception freedom. Note that it is impossible to prove that a variable is within its type until it has been assigned a value, ruling out the candidate invariant property that T is inside its type AC_T.

5.5 Method: For Loop Range

Each SPARK for loop iterator must have a declared type. This type may be constrained by imposing an additional range restriction. For example, consider the source code in Figure 5. The loop iterator I is declared to be of type AR_T and is constrained to be inside a range from L to U. This inspires abstract values which may be expressed as the following candidate invariant.

```
--#assert I>=L and I<=U;
```

Note that the property that loop iterators are within their type, as asserted by the EXAMINER, is usually sufficient for exception freedom proofs. However, the more constrained property found here would likely assist a partial correctness proof.

5.6 Method: Non-looping Code

At the start of a SPARK subprogram an arbitrary variable X will either have its initial value ($x^{\sim}$) or be undefined (*undef*). Following each assignment to X its value will change accordingly. Essentially it is straight forward to propagate the value of variables through non-looping code.

<pre> package FindPackage is subtype AR_T is Integer range 1..10; subtype ARMO_T is Integer range 0..10; type AC_T is range -1000..1000; type A_T is array (AR_T) of AC_T; procedure Find(A: in A_T; L,U: in AR_T; F: in AC_T; R: out ARMO_T); --#derives R from A,L,U,F; end FindPackage; </pre>	<pre> package body FindPackage is procedure Find(A: in A_T; L,U: in AR_T; F: in AC_T; R: out ARMO_T) is begin R:=0; for I in AR_T range L..U loop if A(I)=F then R:=I; exit; end if; end loop; end Find; end FindPackage; </pre>

Fig. 5. Find first index in array, between bounds, containing target

For example, consider the source code shown in Figure 6. At the start of subprogram `Clip` $v = v^\sim$ and $r = \text{undef}$. Entering the `then` branch of the outermost `if` statement yields $r = I.T'First$. Entering the `else` branch enters the innermost `if` statement. The `then` branch yields $r = I.T'Last$ while the `else` branch yields $r = v^\sim$. As either branch of the innermost `if` statement may be taken a disjunction is required giving $(r = I.T'Last) \vee (r = v^\sim)$. This is repeated for the outermost `if` statement giving $(r = I.T'First) \vee ((r = I.T'Last) \vee (r = v^\sim))$. These abstract values may be expressed through the following candidate assertion. Note that as V is an import variable of mode `in` it can not be changed and thus implicitly refers to its initial value.

```
--#assert (R=I.T'First) or ((R=I.T'Last) or (R=V));
```

However, where variables are assigned expressions involving variables, as in $R:=V$, it is often the case that conditional information can be exploited to constrain the abstract values. For example, consider the disjunct $r = v^\sim$ generated above. All that will be known about v is that it is inside its type. As v is declared as an integer this provides a weak constraint on the value of r . Where $R:=V$ is encountered it is known that $\neg(v < I.T'First) \wedge \neg(v > I.T'Last)$. Using inequality reasoning this can be simplified to $(v \geq I.T'First) \wedge (v \leq I.T'Last)$. Replacing $v^\sim$ with this gives a more constrained abstract value following the outermost `if` $r = I.T'First \vee (r = I.T'Last \vee ((r \geq I.T'First) \wedge (r \leq I.T'Last)))$, which can be simplified to $(r \geq I.T'First) \wedge (r \leq I.T'Last)$. This abstract value would be expressed using the following candidate assertion.

```
--#assert (R>=I.T'First and R<=I.T'Last);
```

Consistently performing such reasoning for the general case would become difficult. However, reasonable progress can be made by targeting variables occurring in assigned expressions and employing lightweight inequality reasoning.

<pre> package ClipPackage is subtype I_T is Integer range 1..4; procedure Clip(V: in Integer; R: out I_T); --# derives R from V; end ClipPackage; </pre>	<pre> package body ClipPackage is procedure Clip(V: in Integer; R: out I_T) is begin if V<I_T'First then R:=I_T'First; else if V>I_T'Last then R:=I_T'Last; else R:=V; end if; end if; end Clip; end ClipPackage; </pre>

Fig. 6. Clip from integer to more constrained type

5.7 Method: Looping Code

Looping code presents problems over non-looping code as the abstract values found for variables within the loop should be general enough to describe every iteration. We use *recurrence relations* to describe the value of a variable on an arbitrary iteration. Powerful tools exist to automatically solve certain classes of recurrence relations, *e.g.* MATLAB [31]. Although we have focused on PURRS [33] as we only require a generic recurrence relation solver we are not tied to PURRS.

The transformations applied to a program variable in a loop are expressed as a recurrence relation, *i. e.* the value of a variable on the n^{th} iteration is expressed in terms of variables on previous iterations, usually the $(n-1)^{th}$ iteration. Solving these recurrence relations produces an invariant equating the value of a variable on the n^{th} iteration to an expression involving n . To extract usable properties from the solved recurrence relations it is necessary to eliminate this n .

Solving a variable's recurrence relation may require solutions to other recurrence relations. For this reason the method is separated into sub-methods with the more immediate sub-methods being applied first. Note that the sub-methods are shown below in the order in which they are applied.

Sub-Method: Unchanged: This targets variables that are unchanged inside a loop. Any import variables of mode `in` must remain unchanged throughout a subprogram. These are identified by examining the subprogram's parameter list. Other variables must change inside the subprogram but may remain unchanged inside a loop. These are identified by finding no assignments to the variable inside the loop.

For example, consider the source code shown in Figure 5. By examining the subprogram parameter list it is found that `A`, `L`, `U` and `F` are import variables of

mode in. Recurrence relations are calculated for the remaining variable R . The initial value of r is 0 and no assignments are made to r inside the loop (the only assignment to r takes place on the loop exit). Thus the recurrence relation found for r is $r_{(0)} = 0, r_{(n)} = r_{(n-1)}$ which is solved as $r_{(n)} = 0$. These abstract values may be expressed as the following candidate invariant.

```
--#assert A=A and L=L and U=U and F=F and R=0;
```

Note that the only descriptive property is $R=0$. However, by successfully solving all of the variables the loop analysis of this subprogram can now terminate.

Sub-Method: Constant Change: It is common to modify a variable by a constant value in each iteration of a loop. These are identified by finding that every assignment to a variable occurs outside conditional statements and the assigned expressions only involve this variable and constant values.

For example, in the running example of Figure 1, I is implicitly initialised to 0 and the assignment statement $I := I + 1$ is implicitly seen after each iteration of the loop. This is expressed as the recurrence relation $i_{(0)} = 0, i_{(n)} = i_{(n-1)} + 1$, which is solved as $i_{(n)} = i_{(0)} + n$ and reduced to $i_{(n)} = n$. As this abstract value contains n it can not yet be presented as a candidate invariant.

Sub-Method: Variable Change: A variable may be modified by a variable amount in each iteration of a loop. This can occur in several cases including assigning to a variable inside a conditional statement and assigning a variable an expression which takes different values from an array. In such cases there is not sufficient information to describe the exact value of a variable on the n^{th} iteration. Thus an approximation is made, generalising the search to finding the bounds of all possible values on the n^{th} iteration. We model the extreme end points of these bounds using what we call *extreme recurrence relations*.

For example, in the running example of Figure 1, R is initialised to 0 and the assignment statement $R := R + A(I)$ is seen within the then branch of the *if* statement which is conditional on $A(I) \geq 0$ and $A(I) \leq 100$. The recurrence relation for not entering the *if* statement is $r_{(0)} = 0, r_{(n)} = r_{(n-1)}$, which is solved as $r_{(n)} = 0$. However, the recurrence relation for entering the *if* statement, $r_{(0)} = 0, r_{(n)} = r_{(n-1)} + element(a, i)$, cannot be solved. The problem is that $element(a, i)$ represents a variable change. This problem term is eliminated by generalising to its extreme bounds. Exploiting context information reveals these bounds to be between $r_{(n)} = r_{(n-1)} + 0$ and $r_{(n)} = r_{(n-1)} + 100$. Each of these can be solved and expressed as a range giving the abstract value $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (n * 100))$. Once again, as this abstract value contains n it can not yet be presented as a candidate invariant.

Sub-Method: Counter Variables: During the execution of a loop the value of variables may change. Those variables found to monotonically increase or decrease by one are classified as *counter variables*. Counter variables are very

common and often key to understanding an algorithm, motivating their special classification.

Counter variables can be identified by exploiting the abstract values found by the constant change and variable change sub-methods. For example, in the description of the constant change sub-method it was shown how the abstract value $i_{(n)} = n$ would be found for variable I in the running example of Figure 1. Although the presence of n prevents this from being presented as a candidate invariant, it is straight forward to determine that I is an increasing counter variable initialised at zero (as n can be thought of as an increasing counter variable initialised at zero). There would be little benefit in expressing this property as a program assertion. However, this information can be collected as program properties and be exploited during proof planning. For example, in [23] the counter variable classification is instrumental in progressing an otherwise failed program proof.

Sub-Method: Extracting Properties: Following the loop analysis it is necessary to post-process the solved recurrence relations into new abstract values that eliminate all references to n . This is achieved by replacing n with an expression in terms of the known program variables.

For example, in the running example of Figure 1 the initial abstract values are $i_{(n)} = n$ and $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (n * 100))$. The upper bound of r is expressed in terms of n , however, exploiting $i_{(n)} = n$, n is replaced by i giving the new abstract value $(r_{(n)} \geq 0) \wedge (r_{(n)} \leq (i_{(n)} * 100))$ which may be expressed as the following candidate invariant.

```
--#assert (R>=0) and (R<=(I*100));
```

Note that it is difficult to eliminate n in $i_{(n)} = n$ as r is not described as an equality with n . This failure means that if an invariant property describing I is required then a suitable abstract value discovered from another method must be used instead. Further note that although the loop analysis does not suggest a candidate invariant property for I it does successfully classify it as a counter variable.

5.8 Method: Loop Guards

The loop analysis involves recurrence relations, expressing constraints on variables on the n^{th} iteration. The loop exit could be modeled by constraining the range of n and analysing the recurrence relations associated with variables. However, such an approach can be quite complicated and often unnecessary. Thus we instead consider the loop exit as a special case distinct from the recurrence relation analysis. We focus on finding properties that describe relationships between variables in the loop guard. In particular we check to see if an inequality relationship holds between these variables.

The loop guard is significant as its negation becomes available in the loops iteration VCs. This is the only property that constrains loop iterations and

thus must be exploited to show that monotonically increasing (or decreasing) variables do not increase (or decrease) forever and exceed their legal bounds.

For example, in the running example of Figure 1 it must be proved that $I \leq AR\_T'Last$ is invariant, *i.e.* that I does not exceed the upper bound of its type. This loop implicitly has a loop guard of the form $I = AR\_T'Last$. Thus the induction hypothesis, $i \leq AR\_T'Last$, and the negation of the loop guard, $\neg(i = AR\_T'Last)$, are hypotheses in the loop iteration VCs. Crucially, these can be combined to provide a single inequality constraint $i < AR\_T'Last$. As i is an increasing counter variable the induction conclusion will take the form $i + 1 \leq AR\_T'Last$ which is trivially true given the inequality constraint hypothesis.

However, cases exist where the negation of the loop guard is not sufficiently strong to support such a proof. For example, consider the source code in Figure 4. Assume the following loop invariant, discovered in §5.4, has been added to prove that I and J do not exceed their type.

```
--#assert (I>=ARPO_T'First and I<=ARPO_T'Last) and
--#      (J>=ARPO_T'First and J<=ARPO_T'Last);
```

The loop guard is $I=J$, introducing the hypothesis $\neg(i = j)$ during loop iteration. Knowing that i and j have different values does not constrain the bounds of i and j . The counter variable sub-method reveals that I is an increasing counter variable, J is a decreasing counter variable and that I starts below J . As the loop exits at $I=J$, I can never exceed J , discovering the candidate invariant property $i \leq j$. Adding this invariant property introduces the new hypothesis $i \leq j$ into the loop iteration VCs. This can now be combined with the negation of the loop guard to introduce the inequality constraint hypothesis $i < j$. This is sufficiently strong to prove that both I and J remain within the bounds of their type.

6 Patching Proof Failure

We return to our running example of Figure 1 and proving the RTC VC for the `then`-branch as shown in Figure 2. Our initial proof plan in §4.3 failed with the transitivity critic requesting additional hypotheses corresponding to the schema

$$(r \geq A) \wedge (r \leq B) .$$

This failure activates program analysis of the relevant subprogram generating a collection of program properties. These properties are searched for a suitable candidate invariant constraint on r , guided by the schema above. Such an invariant was discovered in §5.7 and is repeated below.

```
--#assert (R>=0) and (R<=(I*100));
```

Adding this invariant leads to revised RTC VCs, adding the following two hypotheses to the RTC VC shown in Figure 2.

```
H6:    r >= 0 .
H7:    r <= loop__1__i * 100 .
```

7 Planning the Revised VCs

7.1 Loop Invariant Methods

As illustrated above, it is often the case that an invariant must be strengthened before an exception freedom proof can be completed. The stronger invariant properties must be proved. The proof planner tackles loop invariant VCs using the **ripple** method [2,6]. Although space precludes further discussion we note that proving loop invariants via the ripple method has been previously investigated and reported [24,34,25]. For example, in the running example of Figure 1 the strengthened invariant results in two loop invariant VCs, neither of which are automatically discharged by the **SIMPLIFIER**. However, by proof planning using the **ripple** method these proofs can be automated.

7.2 Revisiting the Exception Freedom Methods

We now consider the methods introduced in §4.1 in more detail. Recall that proving exception freedom involves showing that a variable does not violate its legal upper and lower bounds. Let the general value of a variable be denoted by the term $T(V_1, \dots, V_n)$, where V_i , $1 \leq i \leq n$, denote variables. Further let L and U denote the lower and upper constants of a bound. Thus a variable's lower and upper bound checks give rise to goals of the form in (1) and (2) respectively.

$$T(V_1, \dots, V_n) \geq L \quad (1)$$

$$T(V_1, \dots, V_n) \leq U \quad (2)$$

Although we focus on the upper bound case (2), the same general pattern of proof is also applicable to the lower bound case (1). The proof context associated with (2) should contain hypotheses expressing the upper bounds of V_i

$$V_i \leq U_i . \quad (3)$$

Note that the absence of such hypotheses triggers the **transitivity** critic which aims to introduce the missing hypotheses by exploiting our program analysis.

The first step involves the **transitivity** method, reducing (2) to give

$$(T(V_1, \dots, V_n) \leq X_1) \wedge (X_1 \leq U) . \quad (4)$$

The introduction of the meta-variable X_1 prepares the way for the decomposition of $T(V_1, \dots, V_n)$. The second step calls the **decomposition** method to decompose $T(V_1, \dots, V_n)$. This draws upon a collection of substitution axioms for inequalities. The aim of this method is to express the left hand side conjunct of (4) as a conjunction of inequalities of the form

$$V_i \leq X_j . \quad (5)$$

Note that the complete decomposition of $T(V_1, \dots, V_n)$ may require the application of multiple substitution axioms. The third step calls the **fertilise** method to

match the decomposed inequalities against the inequality hypotheses. Matching (5) against (3) instantiates X_j to U_i . This has the effect of instantiating the right hand side conjunct of (4) to give

$$T(U_1, \dots, U_n) \leq U. \quad (6)$$

The fourth and final step involves the elementary method, simplifying (6) such that it can be trivially discharged by the PROOF CHECKER.

The key to the proof plan is the transitivity method as described in Figure 3. Note that the transitivity method introduces a first-order meta-variable into the goal structure that is incrementally instantiated during subsequent proof planning steps. This use of meta-variables is known as *middle-out reasoning* [5] and has been used effectively in guiding proof search within the context of program synthesis [30,35], proof patching [20,21,22] and loop invariant discovery [24,34,25].

7.3 Successful Proof Plan

We now return to proving the `then`-branch of the code in Figure 1 following the patch of an invariant. Once again, we focus on the goal of proving C4

$$(r + \text{element}(a, i)) \leq \text{integer\_last}, \quad (7)$$

noting that the proof context includes the two hypotheses H7 and H5,

$$(r \leq i * 100) \wedge (\text{element}(a, i) \leq 100). \quad (8)$$

The proof planning begins with an application of the transitivity method, rewriting (7) to a conjunction

$$((r + \text{element}(a, i)) \leq X_1) \wedge (X_1 \leq \text{integer\_last}). \quad (9)$$

The decomposition method searches for a substitution axiom involving $\leq$, finding the rewrite rule

$$(W + X) \leq (Y + Z) \Rightarrow (W \leq Y) \wedge (X \leq Z) \quad (10)$$

which is applied to (9) giving

$$((r \leq X_2) \wedge (\text{element}(a, i) \leq X_3)) \wedge ((X_2 + X_3) \leq \text{integer\_last}). \quad (11)$$

Note that as a side-effect of applying (10), X_1 has been instantiated to $X_2 + X_3$ in (11). Given (8), the `fertilise` method applies to the conjuncts on the left hand side of (11) resulting in X_2 and X_3 being instantiated to $i * 100$ and 100 respectively. The remaining goal takes the form

$$((i * 100) + 100) \leq \text{integer\_last}.$$

Given that `integer_last` has a known concrete value this goal is trivial and can be discharged by the elementary method.

8 Related Work

Probably the first system to prove exception freedom was the RUNCHECK verifier [18]. RUNCHECK operated on Pascal programs, employing a number of heuristics to discover invariants and tackling RTC VC proofs with an external theorem prover. One of its heuristics involved the calculation of recurrence relations as *change vectors*, ignoring program context and collecting transformations made to variables. These change vectors were subsequently solved using a few rewrite rules that targeted common patterns. Our approach has a tighter integration between theorem proving and program analysis. In addition, our program analyser solves recurrence relations using a powerful recurrence relation solver tool. Further, our program analysis exploits program context and approximates to ranges where equality solutions can not be found.

The use of recurrence relations in generating loop-invariants was first reported by Elspas *et al* [13] and was also used by Katz and Manna [29]. Although the limits of recurrence relations as a basis for generating loop-invariants are well known [8], they have proved to be very useful for our niche application.

Recently there has been renewed interest in approaches that employ theorem proving to strengthen program development. The focus tends to be on finding errors rather than proving correctness. For example, ESC/JAVA [17] is an extended static checker for Java. Like SPARK, ESC/JAVA requires program annotations. HOUDINI [16] is able to automatically generate many of the annotations required by ESC/JAVA using predicate abstraction.

There exists systems that employ program analysis to pinpoint unfavourable behaviour. These systems are typically formulated inside the abstract interpretation framework [10]. By observing this framework the program analysis will ensure correctness by allowing for approximate results. The most noteworthy systems are MERLE [38] and POLYSPACE [32]. Although these systems do not target proof, their results might be used to assist a formal proof. Rather than use annotations these systems gain constraints on variables by analysing a program in its entirety. This process can be computationally expensive and requires a complete program for input. As our program analysis targets individual subprograms it is fairly cheap to perform and is applicable early in program development. Further, by avoiding the abstract interpretation framework we have the flexibility to implement heuristic based program analysis techniques. As we treat our program analysis as an oracle that guides search for a formal proof, we can adopt this approach without sacrificing correctness.

9 Progress and Future Work

Our NUSPADE tool has been prototyped as separate components. The proof plans presented here have been implemented within the CLAM proof planner [7]. Note that this prototype does not support the extraction of a customised tactic from discovered proof plans. The program analysis has been prototyped in a system with a limited SPARK parser. This is sufficient to explore the program analysis

methods. Work is underway on completing the NUSPADE system. Currently we have developed a suitable proof planning infrastructure in Prolog and are building a stronger program analysis system, exploiting the STRATEGO [37] program transformation tool.

Using our prototype systems we have successfully demonstrated the applicability of our technique on a collection of isolated subprograms. These subprograms are representative of the kinds of subprograms that are seen within a high integrity software system. The next step is to tackle proof of exception freedom for an entire industrial strength high integrity software system. We also envisage a comparative study between our approach and non-theorem proving techniques, such as MERLE and POLYSPACE. It may be found that such systems can be packaged as additional program analysis oracles for use in our approach.

10 Conclusion

Building upon the SPARK toolset, we have developed an approach for increasing the automation of exception freedom proofs. Our approach is formulated within the proof planning framework. Under certain patterns of failure, critics are invoked which in turn appeal to a program analysis oracle. This oracle aims to discover program properties that patch the failed proof, allowing the proof planning to progress.

Our approach demonstrates that program verification can be tackled on more than one front. By integrating the distinct static analysis techniques of proof planning and program analysis a more capable automatic program verification system can be constructed.

Acknowledgements. In particular we would like to thank Peter Amey and Rod Chapman for their support in our research. Thanks also go to Alan Bundy, Jonathan Hammond, Ian O'Neill, Phil Thornley, Benjamin Gorry, Tommy Ingulfson, Julian Richardson and Maria McCann for their feedback and encouragement. The research reported in this paper is supported by EPSRC grant GR/R24081 and is a collaboration with Praxis Critical Systems Ltd.

References

1. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
2. D. Basin and T. Walsh. A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2), 1996.
3. J. Bergeretti and B.A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1), 1985.
4. A. Bundy. The use of explicit plans to guide inductive proofs. In *CADE-9*. Springer-Verlag, 1988.

5. A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In *Proceedings of UK IT*, 1990.
6. A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62, 1993.
7. A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In *10th International Conference on Automated Deduction*, 1990.
8. M. Caplain. Finding invariant assertions for proving programs. In *Proceedings of the International Conference on Reliable Software*, 1975.
9. R. Chapman and P. Amey. Industrial strength exception freedom. In *Proceedings of ACM SigAda*. Addison-Wesley, 2002.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL-4*. ACM, 1977.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL-5*. ACM, 1978.
12. C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Expo (DISCEX)*. IEEE Computer Society Press, 2000.
13. D. Elspas, M.W. Green, K.N. Levitt, and R.J. Waldinger. Research in interactive program-proving techniques. In *SRI*. 1972.
14. A. Ermedahl and J. Gustafsson. Deriving annotations for tight calculation of execution time. In *European Conference on Parallel Processing*, 1997.
15. ESA. Ariane 5 - flight 501 failure. Board of inquiry report, European Space Agency, 1996.
16. C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proceedings of FME*. Springer-Verlag, 2001.
17. C. Flanagan, K. Rustan M. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI*, 2002.
18. S.M. German. Automating proof of the absence of common runtime errors. In *POPL-5*. ACM, 1978.
19. A. Ireland. The use of planning critics in mechanizing inductive proofs. In *International Conference on Logic Programming and Automated Reasoning (LPAR'92)*, LNAI No. 624. Springer-Verlag, 1992.
20. A. Ireland and A. Bundy. Extensions to a generalization critic for inductive proof. In *13th Conference on Automated Deduction*, 1996.
21. A. Ireland and A. Bundy. Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16(1–2), 1996.
22. A. Ireland and A. Bundy. Automatic verification of functions with accumulating parameters. *Journal of Functional Programming: Special Issue on Theorem Proving & Functional Programming*, 9(2), 1999.
23. A. Ireland, B.J. Ellis, and T. Ingulfsen. Invariant patterns for program reasoning. Technical Report HW-MACS-TR-0011, School of Mathematical and Computer Sciences, Heriot-Watt University, 2004. Also to appear in the Proceedings of the Mexican International Conference on Artificial Intelligence 2004 (MICAI-04).
24. A. Ireland and J. Stark. On the automatic discovery of loop invariants. In *Proceedings of the 4th NASA Langley Formal Methods Workshop – NASA Conference Publication 3356*, 1997.
25. A. Ireland and J. Stark. Proof planning for strategy development. *Annals of Mathematics and Artificial Intelligence*, 29(1–4), 2001.
26. ISO. Reference manual for the Ada programming language. ISO/IEC 8652, International Standards Organization, 1995.

27. C.B. Jones. The early search for tractable ways of reasoning about programs. In *IEEE Annals of the History of Computing*. IEEE Computer Society, 2003.
28. S.M. Katz and Z. Manna. A heuristic approach to program verification. In *Proceedings of IJCAI-73*. IJCAI, 1973.
29. S.M. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4), 1976.
30. I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In *Proceedings of the 10th International Conference on Logic Programming*, 1993.
31. MatLab. <http://www.mathworks.com/>.
32. PolySpace-Technologies. <http://www.polyspace.com/>.
33. PURRS: The parma university's recurrence relation solver. <http://www.cs.unipr.it/purrs/>.
34. J. Stark and A. Ireland. Invariant discovery via failed proof attempts. In *Logic-Based Program Synthesis and Transformation*, number 1559 in LNCS. Springer-Verlag, 1998.
35. J. Stark and A. Ireland. Towards automatic imperative program synthesis through proof planning. In *14th IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, 1999.
36. A.M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, UK, 1949.
37. E. Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In *Rewriting Techniques and Applications (RTA)*, LNCS, 2001.
38. L. Whiting and M. Hill. Safety analysis of hawk in flight monitor. In *Workshop on Program Analysis For Software Tools and Engineering*, 1999.

Verifying Controlled Components

Steve Schneider and Helen Treharne

Department of Computer Science, Royal Holloway, University of London

Abstract. Recent work on combining CSP and B has provided ways of describing systems comprised of components described in both B (to express requirements on state) and CSP (to express interactive and controller behaviour). This approach is driven by the desire to exploit existing tool support for both CSP and B, and by the need for compositional proof techniques. This paper is concerned with the theory underpinning the approach, and proves a number of results for the development and verification of systems described using a combination of CSP and B. In particular, new results are obtained for the use of the hiding operator, which is essential for abstraction. The paper provides theorems which enable results obtained (possibly with tools) on the CSP part of the description to be lifted to the combination. Also, a better understanding of the interaction between CSP controllers and B machines in terms of *non-discriminating* and *open* behaviour on channels is introduced, and applied to the deadlock-freedom theorem. The results are illustrated with a toy lift controller running example.

1 Introduction

Morgan's failures/divergences semantics for event systems [Mor90] enables the various CSP semantics to be given to B machines. These CSP semantics allow machines to be treated as CSP components within a concurrent system, and we can combine them with other CSP components using architectural operators such as parallel composition and abstraction.

Recent work [Tre00] has considered the interaction between a particular kind of B machine and a controller written as a (recursive) sequential CSP process. An important requirement of a controller for a machine is that it should invoke machine operations only within their preconditions. Previous results [Tre00] have identified conditions sufficient to guarantee $P \parallel M$ to be divergence-free for a controller P and machine M , which ensures this important property. These results require identification of a *control loop invariant* (CLI) on the state of the B machine M , which must be true on every recursive call. This is established by considering the semantics of the B operations as they are called within the controller, and essentially computing the weakest precondition required to establish the CLI.

In combining communicating B machines, we use a particular architecture [ST02b] to restrict the interaction between components, by ensuring that each B machine interacts only with its own controller. A system will be structured as a collection of B machines $M_1, \dots, M_n$, each with its own CSP controller process

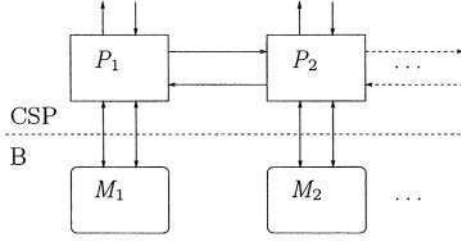


Fig. 1. A CSP and B combined system architecture

$P_1, \dots, P_n$. A *controlled component* is the parallel combination of a controller and its B machine, of the form $P \parallel M$.

Each M_i is under the control of the corresponding P_i , and the P_i 's can also interact with each other. This architecture is illustrated in Figure 1. Interaction across the system can occur only between the CSP processes. This approach enables compositional verification, whereby we are able to verify properties of the entire system by obtaining results about smaller structures within the system. In particular, both CSP and B already have mature tool support which can be used to verify the components.

The model-checker FDR [For97] performs model-checking on systems described in CSP, and is therefore suitable for analysing the controllers, individually and in combination. The paper provides theorems which enable results obtained (possibly with tools) on the CSP part of the description to be lifted to the combination¹. We obtain a number of theorems in the various CSP semantic models.

In practice, we find that it is often the case that a property holds in a combined system for reasons associated with the state within the B components. In this case, the CSP controller descriptions need to be augmented with the relevant state information. This paper also provides theorems which support the required manipulations of CSP controllers. In this paper, we provide informal explanations of the theorems, but for reasons of space cannot include the proofs. Instead, a fuller version of this paper [ST02a] gives proofs of all the theorems and lemmas.

2 Background

2.1 CSP Events

CSP processes are defined in terms of the *events* that they can and cannot do. Processes interact by synchronising on events, and the occurrence of events is atomic. The set of all events is denoted by Σ .

Events may be compound in structure, consisting of a *channel name* and some (possibly none) *data values*. Thus, events have the form $c.v_1 \dots v_n$, where c

¹ The FDR checks discussed in this paper are available at

<http://www.cs.rhul.ac.uk/research/formal/steve/code/lifts.fdr2>

is the channel name associated with the event, and the v_i are data values. The *type* of the channel c is the set of values that can be associated with c to produce events.

For example, if *trans* is a channel name, and $\mathbb{N} \times \mathbb{Z}$ is its type, then events associated with *trans* will be of the form *trans.n.z*, where $n \in \mathbb{N}$ and $z \in \mathbb{Z}$. For example, *trans.3.8* is one such event.

A *partial event*, or (following [Sca98]) *partially completed datatype value* is a channel name together with some values, but not necessarily all. For example, *trans.3* is a partial event. Any channel is a special case of a partial event.

Given a set of partial events PE , we can define the set of events $\{| PE |\}$ which are the completions of events in PE , as follows:

$$\{| PE |\} = \{p.w \mid p \in PE \wedge p.w \in \Sigma\}$$

We use alphabetised CSP, so every process has an alphabet, which is the set of events whose occurrence requires its participation. The alphabet of a process P is denoted $\alpha(P)$. For the purposes of this paper we will require that the alphabet of any process is given by a set of channels C , so that $\alpha(P) = \{| C |\}$.

2.2 CSP Controllers

A controller for a B machine is a particular kind of CSP process. To interact with the B machine, it makes use of control channels which have both input and output, and provide the means for controllers to synchronise with B machines. For each operation $w \leftarrow e(v)$ of a controlled machine with v of type $T_{in}(e)$ and w of type $T_{out}(e)$ there will be a channel e of type $T_{in}(e) \times T_{out}(e)$, so communications on e are of the form $e.v.w$.

Controller descriptions may also include assertions about the values of variables they are using. These are incorporated in CSP either as *blocking assertions* (which block if the assertion is false) or as *diverging assertions* (which diverge if the assertion is false), depending on the role they play in verification.

When we talk about a CSP controller P we mean a process which has a given set of control channels C . The controlled B machine will have exactly $\{| C |\}$ as its alphabet: it can communicate only on channels in C .

Controller syntax. Controllers are generated from the following subset of the CSP syntax, as discussed in [ST02b].

$$\begin{aligned} P ::= & a \rightarrow P \mid c?x \rightarrow P \mid d!v \rightarrow P \mid e!v?x\{E(x)\} \rightarrow P \mid e!v?x\langle E(x) \rangle \rightarrow P \mid \\ & P_1 \sqcap P_2 \mid P_1 \sqcap P_2 \mid \bigcap_{x \in E(x)} P \mid \text{if } b \text{ then } P_1 \text{ else } P_2 \mid S(p) \end{aligned}$$

where a and is is a *synchronisation event*, c is a *communication channel* accepting inputs, d is a communication channel sending output values, e is a *control channel*, x is a data variable, v is a data value, $E(x)$ is a predicate on x (it may be elided, in which case it is considered to be *true*), b is a boolean expression, and $S(p)$ is a process expression.

The process $a \rightarrow P$ is initially prepared to engage in an a event, after which it behaves as P . The input $c?x \rightarrow P$ is prepared to accept any value x along channel c , and then behave as P (whose behaviour can be dependent on x). The output $d!v \rightarrow P$ provides v as output. The operation call $e!v?x\{E(x)\} \rightarrow P$ is an interaction with an underlying B machine: the value v is passed from the process as input to the B operation, and the value x is accepted as output from the B operation. If x meets the condition $E(x)$ then the process behaves as P . If x does not meet the condition then the process diverges. On the other hand, $e!v?x\{E(x)\} \rightarrow P$ only allows $e.v.x$ if $E(x)$, otherwise the event is blocked. Behaviour subsequent to $e.v.x$ is that of P .

The external choice process $P_1 \square P_2$ is initially prepared to behave either as P_1 or as P_2 , and the choice is resolved on occurrence of the first event. Binary and general internal choice are possible, though not used in the example presented here. The conditional choice $\text{if } b \text{ then } P_1 \text{ else } P_2$ behaves as P_1 or P_2 depending on the evaluation of the condition b . The process expression $S(p)$ expresses a recursive call. Finally, processes can be defined using (recursive) definitions of the form $S(p) \triangleq P$.

2.3 CSP Semantic Models

There are three semantic models used in this paper: the *Traces* model, the *Stable Failures* model, and the *Failures/Divergences* model. We introduce the relevant features of them here. Full details of these models can be found in [Ros97,Sch99].

Traces. A *trace* is a finite sequence of events. A sequence tr is a trace of a process P if there is some execution of P in which exactly that sequence of events is performed. The set $traces(P)$ is the set of all possible traces of process P . The traces model for CSP associates a set of traces with every CSP process. If $traces(P) = traces(Q)$ then P and Q are equivalent in the traces model, and we write $P =_T Q$.

Stable Failures. A *stable failure* is a pair (tr, X) consisting of a trace tr and a set of events X . Such a pair is a stable failure of a process P if there is some execution of P on which tr is the sequence of events performed, reaching a state in which all events in X can be refused, and also no internal progress is possible. The set $\mathcal{SF} \llbracket P \rrbracket$ is the set of stable failures of P . The stable failures model for CSP associates a set of stable failures, and a set of traces, with every CSP process. If $\mathcal{SF} \llbracket P \rrbracket = \mathcal{SF} \llbracket Q \rrbracket$ and also $traces(P) = traces(Q)$ then P and Q are equivalent in the stable failures model and we write $P =_{SF} Q$.

Failures and Divergences. A *divergence* is a finite sequence of events tr . Such a sequence is a *divergence* of a process P if it is possible for P to perform an infinite sequence of internal events (such as a livelock loop) on some prefix of tr . The set of divergences of a process P is written $\mathcal{D} \llbracket P \rrbracket$.

A *failure* is a pair (tr, X) consisting of a trace tr and a set of events X . It is a failure of a process P if either tr is a divergence of P (in which case X can be any set), or (tr, X) is a stable failure of P . The set of all possible failures of a process P is written $\mathcal{F} \llbracket P \rrbracket$. If $\mathcal{D} \llbracket P \rrbracket = \mathcal{D} \llbracket Q \rrbracket$ and $\mathcal{F} \llbracket P \rrbracket = \mathcal{F} \llbracket Q \rrbracket$ then P and Q are equivalent in the failures-divergences model, written $P =_{FD} Q$.

The different models are used to analyse CSP systems with respect to different properties. This paper is concerned with the failures-divergences model, which is used to check for liveness properties such as divergence-freedom. If a system description includes the possibility of divergence (for example, if it includes internal events), then it is necessary to use the failures-divergences model to check for divergence-freedom.

An important relationship between the stable failures model and the failures divergences model is that if a process is divergence-free (i.e. its set of divergences is empty), then its failures are the same as its stable failures. This is captured in the following theorem:

Theorem 1. *If $\mathcal{D} \llbracket P \rrbracket = \{\}$, then $\mathcal{F} \llbracket P \rrbracket = \mathcal{SF} \llbracket P \rrbracket$.*

This theorem is useful because it allows us to carry out analysis in the stable failures model, which is generally easier and more efficient, and to establish results which remain valid in the failures-divergences model. For example, once it has been established that a process P is divergence-free, then to check that it is deadlock-free (i.e. that $(tr, \alpha(P))$ cannot be a failure of P for any tr), it is sufficient to check this in the stable failures model (that $(tr, \alpha(P))$ cannot be a stable failure). The model-checker FDR [For97] can carry out divergence-freedom and deadlock-freedom checks mechanically. There are also CSP theorems (for example, Theorem 3 in this paper) for establishing that a process P is divergence-free.

2.4 CSP Semantics for B Machines

Morgan's CSP-style semantics [Mor90] for event systems enables us to define such semantics for B machines. A machine M thus has a set of traces $\mathcal{T} \llbracket M \rrbracket$, a set of failures $\mathcal{F} \llbracket M \rrbracket$, and a set of divergences $\mathcal{D} \llbracket M \rrbracket$. A sequence of operations $\langle e_1, e_2 \dots e_n \rangle$ is a *trace* of M if it can possibly occur. This is true precisely when it is not guaranteed to be blocked, or in other words it is not guaranteed to achieve *false*. In *wp* notation we write $\neg wp(e_1; e_2; \dots; e_n, false)$, or in Abstract Machine Notation $\neg([e_1; e_2; \dots; e_n]false)$. (The empty trace is treated as *skip*). A sequence does not diverge if it is guaranteed to terminate (i.e. establish *true*). Thus, a sequence is a divergence if it is not guaranteed to establish *true*, i.e. $\neg([e_1; e_2; \dots; e_n]true)$. Finally, given a set of events X , each event $e \in X$ is associated with a guard g_e . A sequence with a set of events is a *failure* of M if the sequence is not guaranteed to establish the disjunction of the guards. Thus, $(e_1; e_2; \dots; e_n, X)$ is a failure of M if $\neg([e_1; e_2; \dots; e_n](\bigvee_{e \in X} g_e))$. More details of the semantics of B machines can be found in [Tre00].

Morgan does not give a stable failures semantics for action systems. We will define the stable failures $\mathcal{SF} \llbracket M \rrbracket$ for a machine M in terms of its failures divergences semantics, as follows:

Definition 1. *The stable failures of a B machine are defined as follows:*

$$\mathcal{SF} \llbracket M \rrbracket = \{(tr, X) \mid (tr, X) \in \mathcal{F} \llbracket M \rrbracket \wedge tr \notin \mathcal{D} \llbracket M \rrbracket\}$$

<pre> MACHINE i_Lift VARIABLES i_floor INVARIANT i_floor : NAT INITIALISATION i_floor := 0 OPERATIONS i_inc(nn) = PRE nn : NAT1 THEN i_floor := i_floor + nn END; i_dec = PRE i_floor > 0 THEN i_floor := i_floor - 1 END; bb <-- i_isZero = IF i_floor = 0 THEN bb := TRUE ELSE bb := FALSE END END </pre>	<pre> i_LiftCtrl ≡ i_up?y → i_inc!y → i_LiftCtrl □ i_down?y → i_DOWN(y) □ i_ground → i_LOWER i_DOWN(n) ≡ if n = 0 then i_LiftCtrl else i_isZero?bb → if (bb = TRUE) then i_LiftCtrl else i_dec → i_DOWN(n - 1) i_LOWER ≡ i_isZero?bb → if (bb = TRUE) then i_LiftCtrl else i_dec → i_LOWER </pre>

Fig. 2. A Lift machine *i_Lift* and its controller *i_LiftCtrl*

Observe that with this definition, Theorem 1 also holds for B machines *M*.

We have a technique [Tre00,ST02b], based on control loop invariants, for establishing that a combination $P \parallel M$ is divergence-free. In other words, previous results provide a means to establish that $\mathcal{D} \llbracket P \parallel M \rrbracket = \{\}$. This paper is not concerned with that technique. Rather we are concerned with composing together a number of $P_i \parallel M_i$ pairs once we have established that $\mathcal{D} \llbracket P_i \parallel M_i \rrbracket = \{\}$ for each pair. Hence a number of the theorems in this paper will include an assumption that $\mathcal{D} \llbracket P_i \parallel M_i \rrbracket = \{\}$. The assumption in particular cases can be discharged using the control loop invariant technique.

3 A Motivating Toy Example: A Lift Controller

As motivation for the results presented in this paper, we consider a toy example of a collection of lift machines described in B, controlled by CSP controller processes. We will indicate the use of the theorems presented later in the paper. An individual lift is given in Figure 2. It describes a particular lift, indexed by *i*. We will then go on to define a system consisting of a collection of such lifts.

3.1 Individual Lifts

The Lift machine provides three operations: *i_inc*(*nn*) which moves the lift up *nn* floors, *i_dec* which moves the lift down one floor, and a query operation *i_isZero* which indicates whether or not the lift is on the ground floor.

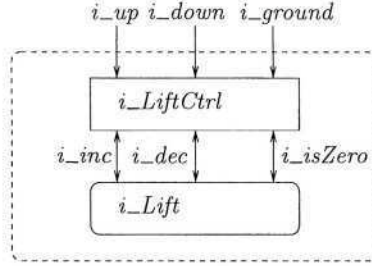


Fig. 3. The controlled lift system

The CSP controller is also given in Figure 2. It interacts with a user through the events *i_up*, *i_down*, and *i_ground*, and controls the lift accordingly:

- on *i_up.y*, it calls *i_inc* and moves the lift up *y* floors.
- on *i_down.y*, it calls *i_dec* *y* times or until it reaches the ground if this is sooner.
- on *i_ground*, it is required to move the lift to the ground floor. To do this, it repeatedly checks (using *i_isZero*) whether the lift is on the ground floor, and if not then it moves the lift down a floor with *i_dec*.

We are firstly interested in each controlled lift combination

$$i\_LiftSys \hat{=} (i\_Lift \parallel i\_LiftCtrl) \setminus \{ | i\_inc, i\_dec, i\_isZero | \}$$

which is pictured in Figure 3. We require as a minimum that this combination is deadlock-free and divergence-free.

These properties are apparent in this simple example. Deadlock-freedom is immediate because the B machine is always willing to engage in any event required by the controller, and the controller itself is either waiting for an interaction from its environment or else ready to call a controller operation. Divergence could arise either (i) from a B operation being called outside its precondition, or (ii) from an infinite sequence of internal events. In the case of (i), the only operation with a non-trivial precondition is *i_dec*, and the controller is constructed so that *i_dec* is only ever called when the lift is not at floor 0. In the case of (ii), the lift will eventually reach the ground floor and so an infinite sequence of calls of *i_dec* cannot occur.

In more complex examples the properties may not be so apparent, and it would be useful to be able to apply analysis tools to carry out model-checking on the combined system. However, no tools currently exist which can analyse a combination of B and CSP descriptions, so instead we analyse the descriptions separately and combine results. In particular, for considering properties such as deadlock and livelock we would aim to apply a tool such as FDR [For97] to the CSP part of the description, and deduce results about the controlled combination. In particular, once it has been established that the controller does not call operations outside their precondition, then the aim is that all deadlocking

$$\begin{array}{ll}
i\_LiftCtrl2(f) \hat{=} & i\_DOWN2(f, n) \hat{=} \\
i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl2(f + y) & \text{if } n = 0 \\
\Box i\_down?y \rightarrow i\_DOWN2(f, y) & \text{then } i\_LiftCtrl2(f) \\
\Box i\_ground \rightarrow i\_LOWER2(f) & \text{else } i\_isZero?bb \\
& \boxed{\{bb = TRUE \Leftrightarrow f = 0\}} \rightarrow \\
i\_LOWER2(f) \hat{=} & \text{if } (bb = TRUE) \\
i\_isZero?bb \boxed{\{bb = TRUE \Leftrightarrow f = 0\}} \rightarrow & \text{then } i\_LiftCtrl2(f) \\
\text{if } (bb = TRUE) & \text{else } i\_dec \rightarrow \\
\text{then } i\_LiftCtrl2(f) & i\_DOWN2(f - 1, n - 1) \\
\text{else } i\_dec \rightarrow i\_LOWER2(f - 1) &
\end{array}$$

Fig. 4. The controller with diverging assertions

and divergent behaviour is essentially contained in the controller and can be identified without further reference to the B machine.

It has previously been established [ST02b] that, under appropriate conditions, the deadlock-freedom of a controller P implies the deadlock-freedom of a controlled combination $P \parallel M$. This result appears in this paper as Theorem 2 in Section 4.

We also establish in this paper (Theorem 3 in Section 5) that, under appropriate conditions, if $P \setminus E$ is divergence-free, then so too is $(P \parallel M) \setminus E$.

These two theorems are exactly what is required. We have only to check that $i\_LiftCtrl$ is deadlock-free to deduce the same for $i\_LiftSys$. And we have only to check that $i\_LiftCtrl \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free to deduce this for $i\_LiftSys$. These are both checks that are easily done using FDR.

However, the second check turns out not to be correct. The description of $i\_LiftCtrl \setminus \{i\_inc, i\_dec, i\_isZero\}$ in fact contains a divergence arising from the infinite sequence $\langle i\_ground, i\_isZero.false, i\_dec, i\_isZero.false, i\_dec, \dots \rangle$ of $i\_LiftCtrl$. It is the machine i_Lift that ensures that this cannot occur — but that machine was not included in the FDR analysis.

The problem is that some of the control flow is dependent on the state information maintained in the B machine, and so the useful theorems we have available are not directly applicable. We need to include the relevant state information in the description of the CSP controller. We do this by introducing a new variable f , and also introducing the expectation that the value *true* will be received on channel i_isZero exactly when $f = 0$. This is included as an assertion, as shown in Figure 4. It is straightforward to show that $i\_LiftCtrl2(0)$ is an appropriate driver for i_Lift (using control loop invariant $f = i\_floor$ which relates the CSP state to the state of the B machine). The proof that $i\_LiftCtrl2(0) \parallel i\_Lift$ has no divergences involves establishing the truth of the assertion for the input bb on i_isZero .

Introducing a diverging assertion means that $i\_LiftCtrl2(0)$ trivially has a divergence (i.e. the behaviour when the assertion is not met), so it is not appropriate to check $i\_LiftCtrl2(0) \setminus \{i\_inc, i\_dec, i\_isZero\}$ for divergence-

$$\begin{array}{ll}
i\_LiftCtrl3(f) \hat{=} & i\_DOWN3(f, n) \hat{=} \\
i\_up?y \rightarrow i\_inc!y \rightarrow i\_LiftCtrl3(f + y) & \text{if } n = 0 \\
\Box i\_down?y \rightarrow i\_DOWN3(f, y) & \text{then } i\_LiftCtrl3(f) \\
\Box i\_ground \rightarrow i\_LOWER3(f) & \text{else } i\_isZero?bb \\
& \quad \boxed{\langle bb = TRUE \Leftrightarrow f = 0 \rangle} \rightarrow \\
i\_LOWER3(f) \hat{=} & \text{if } (bb = TRUE) \\
i\_isZero?bb \boxed{\langle bb = TRUE \Leftrightarrow f = 0 \rangle} \rightarrow & \text{then } i\_LiftCtrl3(f) \\
\text{if } (bb = TRUE) & \text{else } i\_dec \rightarrow \\
\text{then } i\_LiftCtrl3(f) & \quad i\_DOWN3(f - 1, n - 1) \\
\text{else } i\_dec \rightarrow i\_LOWER3(f - 1) &
\end{array}$$

Fig. 5. The controller with blocking assertions

freedom. However, in the context of i_Lift we know the assertion will always be true, so we may replace the diverging assertion by a blocking one, and yield a controller with the same behaviour in the context of i_Lift . The only difference is that this controller blocks rather than diverges when the assertion is false, and since the assertion is never false in the context of i_Lift , the resulting behaviour is the same. This transformation is justified by Corollary 1 (given at the end of Section 5). Thus, we obtain a variant $i\_LiftCtrl3(0)$ of the controller, given in Figure 5, such that $i\_LiftCtrl3(0) \parallel i\_Lift =_{FD} i\_LiftCtrl2(0) \parallel i\_Lift$.

Now we have a transformation of the controller which is divergence-free when the internal events are hidden: $i\_LiftCtrl3(0) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free, and this can be checked using FDR (given a bound on the number of possible consecutive i_up events). So we can conclude that $(i\_LiftCtrl3(0) \parallel i\_Lift) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free.

Now Corollary 1 also allows the assertions of $i\_LiftCtrl2(0)$ to be dropped completely, resulting in a controller whose behaviour does not depend on the value of the parameter f at all, and which is therefore equivalent to $i\_LiftCtrl$. This transformation is discussed in more detail in [ST02a]. We have therefore now established divergence-freeness of the original combination $(i\_LiftCtrl \parallel i\_Lift) \setminus \{i\_inc, i\_dec, i\_isZero\}$.

To sum up: we identified two new controllers which are equivalent in the presence of i_Lift to the original controller $i\_LiftCtrl$, and which are each used in a different part of the proof.

$$i\_LiftCtrl2(0) \parallel i\_Lift =_{FD} i\_LiftCtrl3(0) \parallel i\_Lift =_{FD} i\_LiftCtrl \parallel i\_Lift$$

- The combination $i\_LiftCtrl2(0) \parallel i\_Lift$ can be shown to be divergence-free using techniques from [ST02b].
- $i\_LiftCtrl3(0) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free, and so $(i\_LiftCtrl3(0) \parallel i\_Lift) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free.
- And $i\_LiftCtrl2(0) \parallel i\_Lift$ is equivalent to the original $i\_LiftCtrl \parallel i\_Lift$.

These results together establish the required result: that the original combination $(i\_LiftCtrl \parallel i\_Lift) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is divergence-free. The state

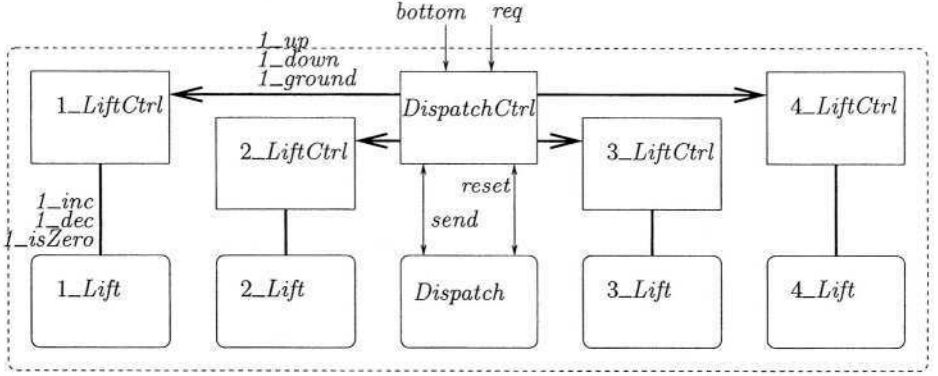


Fig. 6. The complete system *Lifts*

information was introduced into the controller purely to enable the verification to take place, and can be removed once the result has been established.

We also deduce that $(i\_LiftCtrl \parallel i\_Lift) \setminus \{i\_inc, i\_dec, i\_isZero\}$ is deadlock-free. This follows from deadlock-freeness of $i\_LiftCtrl \parallel i\_Lift$.

3.2 A Collection of Lifts

We will now combine the lifts into a single system together with a *Dispatch* and *DispatchCtrl* component which manages requests for lifts from buttons on the various floors. When a request for a lift is made from a particular floor, only one of the lifts needs to be sent. An example architecture made up of four lifts is pictured in Figure 6.

The *Dispatch* machine contains some algorithm for deciding which lift should be sent to a particular floor. It has an operation $ii, nn, dd \leftarrow send(ff)$. On input of the floor ff to send a lift to, it provides as output the lift ii to be sent, the number of floors nn and the direction dd that lift ii will need to travel (as computed by *Dispatch*). *Dispatch* has another operation *reset*, which is called when all lifts return to the ground floor. The particular details of *Dispatch* are not relevant to this example and will not be given here.

The *DispatchCtrl* controller accepts requests along channel *req*: an input $req?x$ is a request for a lift to go to floor x . It makes use of the *Dispatch* machine to decide which lift to allocate, and then sends the appropriate instruction to the relevant lift. The controller can also accept an instruction *bottom* to return all lifts to the ground floor. It is defined as follows:

$$\begin{aligned}
 DispatchCtrl \triangleq & req?x \rightarrow send!x?i?n?d \rightarrow \text{if } d = \text{ascend} \\
 & \quad \text{then } i\_up!n \rightarrow DispatchCtrl \\
 & \quad \text{else } i\_down!n \rightarrow DispatchCtrl \\
 \square & bottom \rightarrow 1\_ground \rightarrow 2\_ground \rightarrow 3\_ground \\
 & \quad \rightarrow 4\_ground \rightarrow reset \rightarrow DispatchCtrl
 \end{aligned}$$

Our overall system is then composed of the controlled lift components $Lifts \triangleq \parallel_{i=1..4} (i\_LiftCtrl \parallel i\_Lift)$ interacting with the $DispatchCtrl \parallel Dispatch$ component, and with all events apart from req and $bottom$ internal:

$$(\parallel_{i=1..4} (i\_LiftCtrl \parallel i\_Lift) \parallel (DispatchCtrl \parallel Dispatch)) \setminus Int$$

$$Int = \bigcup_i \{i\_inc, i\_dec, i\_isZero, i\_up, i\_down, i\_ground\} \cup \{send, reset\}$$

We will see in Section 6 that this system is deadlock-free and divergence-free.

4 Deadlock-Freedom

This section introduces two new properties concerning process behaviour on channels: *open on possible inputs*, and *non-discriminating*. These are the key properties exhibited by B machines and CSP controllers respectively. As we shall see, considering components in terms of these properties enables many of the results from Sections 4 and 5 concerning individual controlled components to be lifted to interacting collections of controlled components in Section 6. They also enable easier proofs of previously established results such as Theorem 2 in this section.

An essential requirement for controlled components is deadlock-freedom. This is easily checked in FDR, but only for processes that are expressed in CSP. Thus, we aim to establish a theorem that allows the deadlock-freedom of $P \parallel M$ to be deduced from deadlock-freedom of P (which can then be checked using FDR).

In general, parallel composition does not preserve deadlock-freedom. Fortunately, in the case of CSP controllers and B machines, we are able to identify conditions which ensure that the processes involved interact on their common channels in a particular way, ensuring that introducing a B machine cannot introduce any new deadlocks. In other words, any deadlocks possible for the controlled component $P \parallel M$ must already have been possible in P .

Open on possible inputs. The required property of the B machine is that it should always be able to accept any input for any operation, and be able to provide some output. The need for this property is precisely why only machines with non-blocking operations are permitted. If a machine meets this property then we will say it is *open* on the particular operations and inputs.

In CSP terms, this is defined formally for CSP processes Q as follows:

Definition 2. A process Q is open on a set of partial events PE if, given any $(tr, X) \in \mathcal{SF} \llbracket Q \rrbracket$ and $e \in PE$, there is some w such that $e.w \notin X$.

This will apply to B machines as follows: given any machine operation $w \leftarrow e(v)$, we would expect the machine to be open on any partial event of the form $e.v_0$, which corresponds to passing the input v_0 to operation e . In other words, there should be some output w_0 which is made available by the machine (and hence does not appear in the refusal set X).

The set of *possible inputs* for a machine will be all those partial events which correspond to operations being called with some input. The events are partial because they do not include the output values.

Definition 3. Given a B machine M with operations $w_i \leftarrow e_i(v_i)$, the set $pi(M)$ of possible inputs for M is defined by

$$pi(M) = \bigcup_i \{e_i.v_i \mid v_i \in T_{in}(e_i)\}$$

Example 1. The set of possible inputs for the machine i_Lift is given in terms of the three operations as follows:

$$pi(i\_Lift) = \{ \mid i\_inc.i \mid i \in \mathbb{Z} \} \cup \{ \mid i\_dec \} \cup \{ \mid i\_isZero \}$$

Observe that in the cases of i_inc and i_dec there are no outputs, so the partial events are in fact complete events. Being open on these events means that they cannot be refused (since their output field is empty). There are two completions of the partial event i_isZero : $i\_isZero.true$ and $i\_isZero.false$. i_Lift being open on this partial event means that at any stage at least one of these completions cannot be refused by i_Lift .

The key property of non-blocking machines is that they will always be open on their possible inputs:

Lemma 1. Any (non-blocking) B machine M is open on $pi(M)$.

This states in CSP semantics terms that any operation call with any input should always produce some result.

Our approach is restricted to *non-blocking* B machines. In other words, operations $w \leftarrow e(v)$ must always be enabled (though they might be called outside their preconditions, which leads to divergence) and on any input they must provide some output. For the purposes of this paper we will henceforth take B machines to be non-blocking.

Non-discriminating controllers. The condition on a controller P is that, whenever it calls an operation of the controlled B machine M , it should be able to accept any output provided by M . We call this property *non-discriminating*, and it can be expressed formally in CSP terms with the following definition:

Definition 4. A CSP process P is non-discriminating on a set of partial events PE if, for any failure $(tr, X) \in \mathcal{SF} \llbracket P \rrbracket$ and subset $CV \subseteq PE$, we have that

$$(\forall c.v \in CV \bullet \exists w \bullet c.v.w \in X) \Rightarrow (tr, X \cup \{ \mid CV \}) \in \mathcal{SF} \llbracket P \rrbracket$$

This definition states that if any event $c.v.w$ can be refused (i.e. appears in the refusal set X), then all the inputs on channel $c.v$ (i.e. outputs from the B machine) could be refused: thus the refusal X can be augmented with $\{ \mid c.v \}$.

Example 2. The control process $i\_LiftCtrl$ is non-discriminating on i_isZero : at any stage, $i\_LiftCtrl$ can either refuse all of $\{| i\_isZero\}$, or else none of it. In terms of the definition, whenever some event from $\{i\_isZero.true, i\_isZero.false\}$ can be refused, then all can be refused.

Observe that $i\_LiftCtrl$ is also non-discriminating on $\{i\_inc.i \mid i \in \mathbb{Z}\}$ and on i_dec . In fact a process will trivially be non-discriminating on complete events.

Controllers which do not include blocking assertions on the control channels are able to accept any output from the associated B machine whenever they call an operation with any particular inputs. Thus, they will be non-discriminating on the possible inputs to the machine. This is expressed by the following lemma:

Lemma 2. *If P is a controller for machine M with no blocking assertions on any channels of M , then P is non-discriminating on the set $\pi(M)$ of M 's possible inputs.*

Observe that this lemma is illustrated by $i\_LiftCtrl$ in Example 2 above.

Establishing Deadlock-freedom. We now have ingredients which are sufficient to deduce deadlock-freedom of $P \parallel Q$ from deadlock-freedom of P . The idea is that the interface between P and Q is defined by a set of partial events PE : P should be non-discriminating on these partial events, and Q should be open on them. We can show that if $P \parallel Q$ can deadlock, then so can P .

If $P \parallel Q$ does have a deadlock state, then all events can be simultaneously refused in that state. For any partial event e , Q is open on e so Q cannot refuse all of $\{| e\}$. Hence P must be refusing some event in $\{| e\}$, and so because P is non-discriminating, P can refuse all of $\{| e\}$. Thus, we find that all events in the interface can be refused by P in this state, and P cannot perform any other events either. Hence P is in a deadlocked state.

Consider this reasoning in the context of a controlled component. Consider a state of $P \parallel M$. If P in this state is not deadlocked, then either

1. P is ready to perform an event outside $\alpha(M)$. In this case, M cannot prevent that event, and the combination $P \parallel M$ is ready to perform the event, and hence is not deadlocked; or
2. P is ready to perform an interaction with M . In this case, it is an operation call c with some input v . P is ready to accept any output from this operation call, since it is non-discriminating on $c.v$. M is ready to provide an output w in response to $c.v$, since it is open on $c.v$. Hence, the combination $P \parallel M$ is ready to perform $c.v.w$, and so is not in a deadlocked state.

The lemma that this reasoning establishes is the following:

Lemma 3. *If*

1. P is non-discriminating on a set of partial events PE ; and
2. Q is open on PE ; and
3. $\alpha(Q) = \{| PE\}$;

then: if P is deadlock-free in the stable failures model, then so too is $P \parallel Q$.

For a particular controlled component $P \parallel M$, we already have the conditions for Lemma 3: P is non-discriminating on $pi(M)$ (from Lemma 2); M is open on $pi(M)$ (from Lemma 1); and $\alpha(M) = \{ \mid pi(M) \mid \}$.

Finally, we obtain the following theorem for controlled components:

Theorem 2. *If P is a CSP controller for M with no blocking assertions on any channels of M , and P is deadlock-free in the stable failures model, then $P \parallel M$ is deadlock-free in the stable failures model.*

This theorem is exactly what is required to establish deadlock-freedom of $P \parallel M$ from deadlock-freedom of P . In fact a direct proof of this theorem in terms of the CSP semantics has previously been presented, in [ST02b]. However, we find the identification of the properties non-discriminating and open yields more understanding as to why the theorem works and allows an easier proof of Theorem 2 and others.

Example 3. For example, consider the combination $i\_LiftCtrl \parallel i\_Lift$, in a state after some trace tr , in which $\{i\_isZero.true, i\_isZero.false\}$ is refused. We know that i_Lift is open on $\{ \mid i\_isZero \mid \}$, so it cannot refuse the whole set $\{i\_isZero.true, i\_isZero.false\}$. Since the parallel combination does refuse that whole set, it must be that $i\_LiftCtrl$ is refusing at least one of $i\_isZero.true$, $i\_isZero.false$. But $i\_LiftCtrl$ is non-discriminating on i_isZero , so this means that it can itself refuse the whole set $\{ \mid i\_isZero \mid \}$.

The same reasoning applies to all partial events in the interface between $i\_LiftCtrl$ and i_Lift . Thus, if $i\_LiftCtrl \parallel i\_Lift$ could reach a deadlock state, then all events in the interface would be refused by $i\_LiftCtrl \parallel i\_Lift$, and so they could also be refused purely by $i\_LiftCtrl$. Thus, $i\_LiftCtrl$ would also have a deadlock state.

As observed previously, $i\_LiftCtrl$ is deadlock-free. Hence Theorem 2 allows us to deduce that $i\_LiftCtrl \parallel i\_Lift$ is deadlock-free.

5 Restricting Events to Prevent Divergence

The use of abstraction is essential in the compositional development of large systems. We will therefore generally need to hide control channels within controlled components. In the lift component example in Section 3, the channels i_inc , i_dec , and i_isZero are hidden, leaving i_up , i_down , and i_ground as the only external channels.

Since hiding has the potential to introduce divergence, we need to be able to establish when this does not occur. In particular, it would be useful to be able to check divergence-freedom of a controller $P \setminus C$ using FDR, and to be able to deduce divergence-freedom of the controlled component $(P \parallel M) \setminus C$.

The following theorem on CSP processes P and Q gives such a condition:

Theorem 3. *If $P \parallel Q$ is divergence-free, and $C \subseteq \alpha(P)$, and $P \setminus C$ is divergence-free, then $(P \parallel Q) \setminus C$ is divergence-free.*

This is immediately applicable to controlled components (where the machine M is considered as the process Q) since $C \subseteq \alpha(P)$ as a consequence of our architecture. Thus, divergence-freedom of $(P \parallel M) \setminus C$ follows directly from divergence-freedom of $P \setminus C$.

However, in practice it will often be the case that $P \setminus C$ turns out not to be divergence-free, even if $(P \parallel M) \setminus C$ is. For instance, in the lift example we found that $i\_LiftCtrl \setminus \{inc, dec, isZero\}$ was not divergence-free, and instead we had to transform the controller description to $i\_LiftCtrl3(0)$ in order to obtain a controller such that $i\_LiftCtrl3(0) \setminus \{inc, dec, isZero\}$ is divergence-free. So it is necessary to identify theorems which justify such transformations.

Our approach is to identify behaviours of controller P which cannot occur in the context of the machine M under control. We then aim to find P' such that

1. P' is the same as P except (possibly) on the behaviours that have been identified, and
2. $P' \setminus C$ is divergence-free

Thus, $P' \parallel M$ will be the same as $P \parallel M$. We are assuming that $P \parallel M$ has previously been shown to be divergence-free: that P is an appropriate controller for M . Theorem 3 applied to P' yields that $(P' \parallel M) \setminus C$ is divergence-free, and hence $(P \parallel M) \setminus C$ is divergence-free.

This was the approach taken in the lift example. The relevant behaviour that cannot occur in the context of i_Lift is the output of *false* from *isZero* when the lift is at the ground floor. This behaviour is blocked in $i\_LiftCtrl3(0)$. However, $i\_LiftCtrl3(0)$ is the same as $i\_LiftCtrl$ for all behaviours that are possible in parallel with i_Lift .

The way we identify traces that cannot occur is to require divergence whenever they do occur, and then look for divergences. If we are concerned with a set of traces $T \subseteq A^*$, then we can express this by defining a new process $DIV_A(T)$ which behaves as RUN_A except that it diverges on any trace in T :

$$\begin{aligned}\mathcal{F} \llbracket DIV_A(T) \rrbracket &= \{(tr, \{\}) \mid tr \in A^*\} \cup \{(tr \frown tr', X) \mid tr \in T \wedge tr' \in A^* \wedge X \subseteq A\} \\ \mathcal{D} \llbracket DIV_A(T) \rrbracket &= \{tr \frown tr' \mid tr \in T \wedge tr' \in A^*\}\end{aligned}$$

Observe that $DIV_A(\{\}) =_{FD} RUN_A$ and $DIV_A(A^*) =_{FD} DIV_A$.

The process $DIV_A(T)$ can then be used to mask behaviour in a process P . The process $P \parallel DIV_A(T)$ behaves exactly as P , except that whenever a trace in T is performed then it diverges. Thus, if $P \parallel DIV_A(T) =_{FD} P' \parallel DIV_A(T)$, then P and P' have the same behaviour except possibly with regard to traces in T , which are masked by the introduction of divergence.

The following theorem allows a process P to be replaced by an alternative process P' in the context of another process Q . In particular, if P does not diverge in the context of Q (i.e. $P \parallel Q$ is divergence-free), and P' is the same as P except on divergent traces of P , then P and P' have the same executions when executed in parallel with Q (since none of P 's divergent traces will be performed).

Theorem 4. *If P , P' and Q are such that*

1. $P \parallel Q$ is divergence-free,
2. $P =_{FD} P' \parallel DIV_{\alpha(P)}(\mathcal{D}[\![P]\!])$
3. $\alpha(P) = \alpha(P')$

then $P \parallel Q =_{FD} P' \parallel Q$.

This states that if P' is different to P only with respect to where P diverges, and $P \parallel Q$ does not diverge, then P and P' behave the same in the context of Q . This follows because if $P \parallel Q$ does not diverge, then none of the traces of P which lead to divergence are possible when executing in parallel with Q . Since P' is exactly the same as P except for these traces, and Q prevents such traces from occurring, it follows that $P' \parallel Q$ is the same as $P \parallel Q$.

Example 4. As an example to illustrate Theorem 4, consider the following processes. P and P' have alphabet $A = \{a, b, c\}$, and Q has alphabet $\{a, b\}$.

$$\begin{aligned} P &\triangleq (a \rightarrow (b \rightarrow DIV_A \square a \rightarrow c \rightarrow P)) \\ P' &\triangleq (a \rightarrow (b \rightarrow c \rightarrow P' \square a \rightarrow c \rightarrow P')) \\ Q &\triangleq (a \rightarrow a \rightarrow Q) \square (b \rightarrow STOP) \end{aligned}$$

- Firstly, we see that $P \parallel Q$ can only ever perform a and c events, and is deadlock-free. In particular, the process Q prevents P from performing the b event, the only event that can lead to divergence, since there is no point at which P and Q can agree to perform b .
- The behaviour of P' after b occurs is different to that of P (which is divergent), but if b does not occur then P and P' behave the same. Thus, P and P' are the same except on the divergences of P .
- Finally, note that P and P' have the same alphabet.

Thus, we can conclude that $P \parallel Q =_{FD} P' \parallel Q$.

The reason this result is useful is because it supports the introduction and manipulation of assertions on the control channels. If we introduce a divergent assertion on a control channel between P and M , and we then establish that $P \parallel M$ is divergence-free (using CLI techniques), then we can alter the behaviour of P when the assertion is false (in which case P diverges) and obtain a related controller P' which matches P outside P 's divergences, and for which $P \parallel M =_{FD} P' \parallel M$. The aim is to obtain a controller P' in this way for which $P' \setminus C$ is divergence-free.

The next lemma lists some ways in which diverging assertions within a controller can be transformed.

Lemma 4. *If a controller P' is obtained from controller P by replacing clauses of the form $e!v?x\{E(x)\} \rightarrow R(x)$ with one of:*

1. $e!v?x\{E'(x)\} \rightarrow R(x)$ where $\forall x. E(x) \Rightarrow E'(x)$
2. $e!v?x \rightarrow \text{if } E(x) \text{ then } R(x) \text{ else } Q(x)$
3. $e!v?x \rightarrow R(x)$
4. $e!v?x\langle E(x) \rangle \rightarrow R(x)$

then $P =_{FD} P' \parallel DIV_{\alpha(P)}(\mathcal{D}[\![P]\!])$

Thus, we obtain the following corollary for controlled components:

Corollary 1. *If $P \parallel M$ is divergence-free, then behaviour in P following an input which fails a diverging assertion can be changed in accordance with Lemma 4 without affecting the behaviour of the parallel combination.*

This means that diverging assertions in P , once they have been discharged in a context M , can be replaced with blocking assertions, or else removed completely. This is precisely the justification for the transformation of $i\_LiftCtrl2(i)$ to $i\_LiftCtrl3(i)$: in the context of i_Lift , $i\_LiftCtrl2(0)$ does not diverge.

6 Parallel Combinations of Controlled Components

All the results of the previous sections have been presented as applying to a single CSP controller process P in parallel with a single B machine M . However, systems we are generally concerned with (such as the combination of lifts) have the form $\parallel_i (P_i \parallel M_i)$, as illustrated in Figure 1. Many of the results we have obtained for a single controlled component can be lifted to combinations of components, and we will consider some of these in this section.

Divergence-freedom. Firstly, we consider divergence-freedom. It is straightforward to establish divergence-freedom of a combined system, using the following theorem from [ST02b]:

Theorem 5. *If $P_i \parallel M_i$ are divergence-free for each i , then $\parallel_i (P_i \parallel M_i)$ is divergence-free.*

This follows immediately from the semantics for parallel composition, which preserves divergence-freedom. Thus, we need only establish divergence-freedom for the component pairs, and the result follows.

Example 5. In the parallel lift system, since each of the controlled lift components is divergence-free, and since we are given that the controlled dispatcher component is divergence-free, it follows that the overall parallel combination of all the components of the multiple lift system is divergence-free.

Establishing deadlock-freedom. Associativity and commutativity of the parallel operator means that we can group the controller processes together and the machines together, rearranging the parallel composition as follows:

$$\parallel_i (P_i \parallel M_i) =_{FD} (\parallel_i P_i) \parallel (\parallel_i M_i)$$

Now we can consider $(\parallel_i P_i)$ as a CSP process, and $(\parallel_i M_i)$ as another CSP process; and we are concerned with the parallel combination of these two processes.

The reason for grouping the components in this way is that the properties ‘non-discriminating’ and ‘open’ are preserved by parallel composition in CSP.

We can thus obtain the following two lemmas:

Lemma 5. *If P_i is a collection of controllers for machines M_i respectively, where each P_i has no blocking assertions on any channels of its associated M_i , then $\parallel_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.*

Lemma 6. *Any collection of (non-blocking) B machines M_i has that $\parallel_i M_i$ is open on $\bigcup_i (pi(M_i))$.*

Lemma 6 states that if each machine is able to engage in any of its operations, then the parallel combination of all the machines is able to engage in any of the operations of any of its machines.

These two lemmas mean that the conditions for Lemma 3 are met for controllers with no blocking assertions:

1. $\parallel_i P_i$ is non-discriminating on the set $\bigcup_i (pi(M_i))$.
2. $\parallel_i M_i$ is open on $\bigcup_i (pi(M_i))$.
3. $\alpha(\parallel_i M_i) = \{ \mid \bigcup_i (pi(M_i)) \mid \}$.

This means that Lemma 3 is directly applicable to a collection of parallel controlled components, in which deadlock-freedom of the overall parallel combination follows from deadlock-freedom of the combination of controllers.

Theorem 6. *Given a collection of CSP controllers P_i and corresponding controlled machines M_i , such that no controller has any blocking assertions on the control channels: then if $\parallel_i P_i$ is deadlock-free in the stable failures model, then so too is $\parallel_i (P_i \parallel M_i)$.*

In the example lift system, we have therefore only to check that

$$(\parallel_{i=1..4} i\_LiftCtrl) \parallel DispatchCtrl$$

is deadlock-free (which is easily shown) to deduce this for the complete system.

Divergence-freedom of Lift System. We are really concerned with divergence-freedom of

$$(\parallel_{i=1..4} (i\_LiftCtrl \parallel i\_Lift)) \parallel (DispatchCtrl \parallel Dispatch)) \setminus Int$$

Theorem 3 is the appropriate theorem to apply here. We need to split the system into P and Q such that $P \parallel Q$ is divergence-free, and $P \setminus C$ is divergence-free. The natural approach would take P as the combination of CSP controllers, and Q as the combination of B machines; verification could indeed be established by introducing assertions into the controllers along the lines of Section 3.

However, we have already established the individual lifts are divergence-free, so we can re-use this result by splitting the system differently, as pictured in

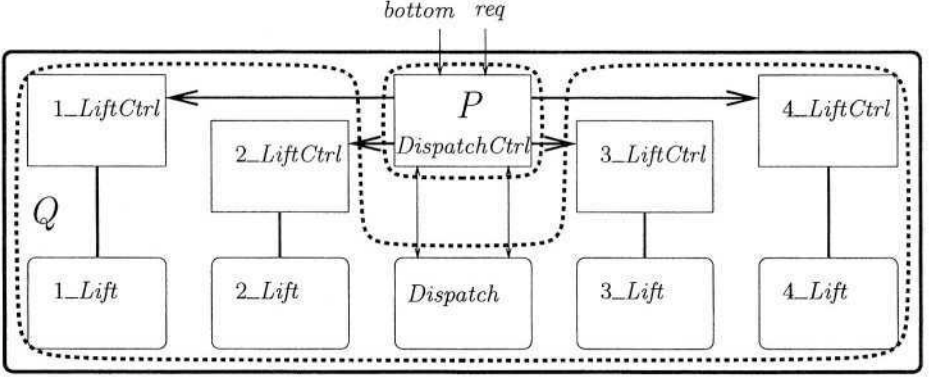


Fig. 7. Splitting the system into P and Q to verify divergence-freeness

Figure 7. P is $DispatchCtrl$, Q is the rest of the system, and C is the interface between P and Q :

$$P \hat{=} DispatchCtrl$$

$$Q \hat{=} \parallel_i i\_LiftSys \parallel Dispatch$$

$$C = \bigcup_i \{ | i\_up, i\_down, i\_ground | \} \cup \{ | send, reset | \}$$

We can check the conditions for Theorem 3:

1. Each $i\_LiftSys$ is divergence-free (established earlier); also $DispatchCtrl \parallel Dispatch$ is divergence-free. This yields that the parallel combination $P \parallel Q = \parallel_i i\_LiftSys \parallel Dispatch \parallel DispatchCtrl$ is divergence-free (since divergence-freeness is preserved by parallel composition).
2. $C \subseteq \alpha(P)$
3. $P \setminus C$ is divergence-free. (This is easily checked with FDR.)

Thus $Lifts \hat{=} (P \parallel Q) \setminus C$ is divergence-free.

7 Discussion

This paper has been concerned with providing the CSP underpinnings for developing controlled components consisting of B machines controlled by CSP controllers under a particular architecture. The work builds on the *control loop invariant* method for verifying individual controlled components in the context of the B Method, and develops results for combining such verified components.

All of the results presented in this paper have been developed using the CSP semantics of all the component processes. The emphasis has been on obtaining compositional results which enable existing CSP verification methods and tools to apply to our combined systems. These results enable a particular strategy for verification: transform system descriptions to equivalent forms which are

amenable to CSP checking. In the simplest case, if the combination $P \parallel M$ is equivalent to $P' \parallel M$, and properties of $P' \parallel M$ can be established by analysing P' (with CSP tools), then those same properties can be deduced for $P \parallel M$. So our approach is to transform a controller P to a process P' which behaves the same way in the context of M .

Transforming system descriptions to enable pure CSP analysis may involve the introduction of state information within the CSP controller descriptions, so that the behaviour in the context of the underlying B machine is not affected. In this paper we have illustrated the use of this technique.

Ongoing work [ST02a] has obtained further results for this framework. Firstly, it is often the case that controlled components are only correct in the context of the rest of the system. In this situation we will need to introduce assertions on the channels between CSP controllers, in order to establish divergence-freedom of the individual controlled components. Treating assertions as blocking or diverging in particular cases is a delicate issue and depends on the particular verification under consideration. We have developed theorems [ST02a] which justify the use of particular kinds of assertions. Secondly, we have results (whose proofs use the new notions of ‘non-discriminating’ and ‘open’) concerning refinement in the stable failures model: if $SPEC \sqsubseteq P \setminus \alpha(M)$ then $SPEC \sqsubseteq (P \parallel M) \setminus \alpha(M)$ under the appropriate conditions. This enables specified properties to be verified of combined systems. These results have been applied to a Bounded Retransmission Protocol [EST03] for buffer-style properties, and in the Bank case study [TSB03].

The toy examples and the case studies carried out to date have provided some experience in the way in which state, and conditions on it, are introduced into the CSP controllers. The necessary state emerges during the verification process in response to FDR checks that fail. Often it is some part of the B state that is simply duplicated in the CSP (as in our toy lift example) in order to enable verification. However, it is too early to identify patterns that may arise in this process (let alone automate it), and more case studies are being pursued.

Scalability of the approach is also a significant issue. Compositionality is a key ingredient of scalability, and it will be important to continue to identify ways in which both requirements and components can each be decomposed to minimise the amount of state required in each verification. This is the subject of ongoing research. In particular, the verification of a controlled component $P \parallel M$ against a collection of requirements might require different state to be introduced into P for each requirement, as was found in the Bounded Retransmission Protocol case study [EST03]. This is better than including all the required state for all of the required properties at once, which could result in duplicating all of the B state in the CSP controller.

There are several other approaches to combining a process-style controller with a state-based system description (e.g. [But00,FL03,WC01,SD01]). The approach closest to ours is Butler’s *csp2B* tool [But00], which allows a CSP process to be conjoined to a B machine in a way which corresponds to a controller for an underlying machine. However, none of the other approaches exploit the semantic models for CSP in the way presented here. The ability to develop theory and tap into existing tool support on both the concurrency side and the state-based side

is an important driver of the approach presented in this paper, and originally motivated the choices of CSP and B as the methods we chose to integrate.

Acknowledgements. Thanks are due to Neil Evans, Susan Stepney, Fiona Polack and Régine Laleau for discussions on this work, and also to Neil Evans and to the anonymous reviewers for their useful comments.

References

- [But00] M. Butler. csp2B: A practical approach to combining CSP and B. *Formal Aspects of Computing*, 12, 2000.
- [EST03] N. Evans, S. A. Schneider, and H. E. Treharne. Investigating a file transmission protocol using CSP and B. In *proceedings of ST.EVE workshop*, 2003.
- [FL03] M. Frappier and R. Laleau. Proving event ordering properties for information systems. In *ZB2003*, 2003.
- [For97] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 Manual*, 1997.
- [Mor90] C. C. Morgan. Of wp and CSP. In W.H.J. Feijen, A. J. M. van Gasteren, D. Gries, and J. Misra, editors, *Beauty is our Business: a birthday salute to Edsger J. Dijkstra*. Springer-Verlag, 1990.
- [Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
- [Sca98] B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP*. D. Phil thesis, Oxford University, 1998.
- [Sch99] S.A. Schneider. *Concurrent and Real-time Systems: The CSP approach*. Wiley, 1999.
- [SD01] G. Smith and J. Derrick. Specification, refinement and verification of concurrent systems - an integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3), 2001.
- [ST02a] S. Schneider and H. Treharne. CSP theorems for communicating B machines. Technical Report CSD-TR-02-05, Royal Holloway, University of London, 2002.
- [ST02b] S.A. Schneider and H.E. Treharne. Communicating B machines. In *ZB2002*, volume LNCS 2272, 2002.
- [Tre00] H. E. Treharne. *Combining control executives and software specifications*. PhD thesis, Royal Holloway, University of London, 2000.
- [TSB03] H.E. Treharne, S.A. Schneider, and M. Bramble. Combining specifications using communication. In *ZB2003*, 2003.
- [WC01] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In *5th Irish Workshop on Formal Methods*, 2001.

Efficient CSP_Z Data Abstraction

Adalberto Farias, Alexandre Mota, and Augusto Sampaio

Federal University of Pernambuco

Informatics Center

P.O.Box 7851, Recife – Brazil

{acf,acm,acas}@cin.ufpe.br

Abstract. This paper proposes an algorithm for abstracting infinite state CSP_Z —formal combination of CSP (behavioural part) and Z (data part)—processes, with the aim of model checking. Differently from previous work, where CSP_Z process abstraction is achieved by investigating only its data part, the current approach abstracts by exploring the whole CSP_Z process. In this way we obtain a faster abstraction algorithm in general, more specific data abstractions, and a wider class of infinite state CSP_Z processes to deal with.

Keywords: Integrated formalism, CSP_Z , specification, verification, model checking, data abstraction, tool support, Java.

1 Introduction

Integration of formal languages has emerged as an elegant and effective framework to describe distinct aspects, such as behaviour, data, time and mobility, of systems simultaneously. In particular, process algebras like CSP [13,23] or CCS [19] are well-suited to model behavioural aspects whereas model-based languages like Z [25] or VDM [16] are more adequate to describe data aspects. Thus, combining languages with these complementary purposes originates a new unifying language, which is suitable to specify behavioural and data structures aspects in a single environment.

The literature reveals some integrated notations: CSP_Z [7,8] (integration of CSP and Z), CSP_{OZ} [9] (combination of CSP and Object-Z [3]), ZCCS [11] (combination of Z and CCS), MOSCA [27] (integration of CCS and VDM), LOTOS [17] (integration of CCS and ACT ONE [4]) and Circus [29] (a language based on CSP, Z, and the *Unifying Theories of Programming* [14]) among others. The present work concentrates on CSP_Z , although the techniques proposed here can, in principle, be adapted to other integrations of process algebra and model-based formalisms.

Investigating such formalisms is generally addressed by a compositional approach, where each constituent language is analysed independently and, latterly, combined. However, applying specifically designed approaches to the integrated formalism might potentially yield more fruitful results. When the language CSP_Z firstly emerged [7], its analysis was naturally thought of as applying model checking to CSP, theorem proving to Z and finally promoting the partial results to

conclude the CSP_Z overall properties. However, work reported in [21] has provided a more general analysis framework for CSP_Z . It has presented an alternative way of applying model checking directly on CSP_Z based on the reuse of resources available for CSP.

Unfortunately, due to the *state explosion* problem, model checking CSP_Z is limited since infinite state CSP_Z processes come out very naturally due to the level of abstraction provided by the Z language. But, as it is well-known for the model checking community, this is a general limitation of model checking. Many auxiliary techniques have been proposed [1,12,18,20,23,26,31,32] to overcome such a limitation: elimination of symmetry, abstraction, symbolic execution, partial order methods, data independence and integration of tools (such as model checkers and theorem provers).

In this paper we present an abstraction approach for CSP_Z aiming at applying model checking to infinite state systems. The novelty is to generate the abstraction from the whole CSP_Z process and not only from the Z part, as reported in previous work [20,22]. The advantages of our approach range over a faster algorithm, more specific data abstractions, and dealing with a wider class of infinite state CSP_Z processes. Furthermore, as in [22] we mechanise the approach, but with a more robust and stable prototype written in Java [15].

The following section provides an overview of CSP_Z . A brief discussion on how model checking can be applied to CSP_Z is given in Section 3. CSP_Z data abstraction is explained in Section 4. Section 5 introduces the algorithm which implements our data abstraction approach. Finally, Section 6 presents our conclusions and future work.

2 CSP_Z Overview

The specification language CSP_Z [7,8] is a formal integration of the process algebra CSP, used to describe behaviour (the behavioural part), and the model-based language Z , used to describe data structures (the data part).

This is a typical example of an integrated formalism where simultaneous systems aspects are modelled in an orthogonal fashion. That is, we can pay attention to the behavioural part of the system and its data part separately.

The semantics of CSP_Z is defined in terms of the semantics of CSP [13,23]; Z is given a CSP semantics. The language CSP has three semantical models: the *traces* ($\mathcal{T}$) model which is concerned with what a process can do (sequences of visible events), the *failures* ($\mathcal{F}$) model which also captures what a process cannot do (the refusals), and the *failures-divergences* ($\mathcal{FD}$) model which deals with divergent behaviour (infinite looping of hidden events); the *failures-divergences* model is the standard semantical model of CSP.

In the following we present a simple CSP_Z specification. Its purpose is to illustrate the basic elements of a CSP_Z language as well as introduce a trivial infinite state CSP_Z process which cannot be analysed using model checking.

Example 1. An infinite clock.

```

spec Clock
  chan tick, tack                                (channel declaration – CSP part)
  main = tick → tack → main                      (main process – CSP part)
  State  $\hat{=}$  [ $c : \mathbb{N}$ ]                                (state definition – Z part)
  Init  $\hat{=}$  [State' |  $c' = 0$ ]                        (initialisation – Z part)
  com_tick  $\hat{=}$  [ $\Delta$ State |  $c' = c + 1$ ]            (operation – Z part)
  com_tack  $\hat{=}$  [ $\Delta$ State |  $c' = c + 1$ ]            (operation – Z part)
end_spec Clock

```

From Example 1 we can see that a CSP_Z specification is delimited by the keywords **spec** and **end_spec**, followed by the name of the process. Inside we have the CSP part—where we can find the channel declarations (*tick* and *tack*) and a process *main*—followed by the Z part—where we can find the schemas *State*, *Init*, *com_tick*, and *com_tack*.

In general, the CSP part contains channel declarations (the keyword *chan* declares visible channels and *lchan* invisible ones), where visible channels are those visible outside the **spec/end_spec** scope whereas local channels are invisible outside such a scope. Additionally, the CSP part has definitions of processes, where the process *main* is required to define the starting point of the behavioural part. Complementarily, the Z part contains schemas defining the state space (keyword *State*), initialisation (keyword *Init*) and operations (keyword *com_* followed by the name of a channel or any Z valid schema name).

Z schemas of the form *com_e*, for a channel name *e*, are executed when *e* occurs as well as refuses *e* when its precondition is not valid. This defines the blocking semantics of CSP_Z in the sense that the CSP part depends on the Z one and vice-versa, and that the Z part does not introduce divergence. Non-*com_* schemas are only performed by means of *com_* schemas using schema operators.

Finally, a CSP_Z process behaves as follows. At the initialisation, the Z part executes the schema *Init*. The CSP part only performs an invisible action (τ). Afterwards, both parts progress in a complete synchronised behaviour: the CSP part performs an event *ev* if and only if its corresponding schema *com_ev* is enabled. Furthermore, if a CSP_Z process performs a trace, then its CSP part performs the same trace and its Z part executes a composition of the corresponding schemas. For example, suppose the trace $\langle \text{tick}, \text{tack} \rangle$ was observed. This implies that the CSP part performed it, and the Z part executed the composition *com_tick* ; *com_tack*.

This way a CSP_Z process can also be represented by a *Labelled Transition System* (LTS). Such an LTS is a directed graph whose nodes contain the state, and the transitions represent an event occurrence (filled arrow) and its corresponding schema execution (dotted arrow). Figure 1 presents this representation for the process *Clock*, whose behaviour is briefly described as follows.

First, the initialisation schema occurs assigning 0 to the state variable $c : \mathbb{N}$ ($c' = 0$). After that, the *main* process only evolves when the event *tick* occurs

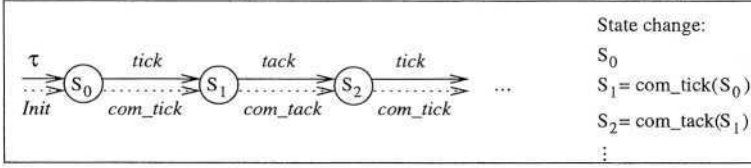


Fig. 1. LTS for a CSP_Z process

and the precondition of *com_tick* is valid. Since the precondition $\text{pre } \textit{com\_tick}$ is valid, the event *tick* can occur, in which case the state variable c is incremented by 1 ($c' = c + 1$). The same applies to the next event (*tack*) and so on. That is, *Clock* is a process which performs the infinite trace $\langle \textit{tick}, \textit{tack}, \textit{tick}, \textit{tack}, \dots \rangle$ while the state variable c assumes the values 0, 1, ..., accordingly. Therefore, this is a typical process which cannot be analysed directly via model checking.

Regarding refinement, the CSP_Z language was defined in such a way that its refinement can be obtained either by CSP or by Z refinements [9]. But to achieve that, all data manipulation must be confined to the Z part and the CSP part is responsible only for control flow.

3 Model Checking CSP_Z

Analysing an integrated language is not so simple because the constituent parts normally influence each other; they are not semantically independent although it is often natural to have a clear syntactical separation among them. Thus, sometimes an independent analysis of each part may not give enough information to conclude some desired property for the entire specification. On the other hand, an analysis strategy specifically tailored to the integrated language potentially yields more interesting results while requires a considerable effort to be conceived.

As a tradeoff between these approaches, one can homogenise the constituent parts in such a way that analysis for the whole integration can be obtained from the analysis of the homogeneous language. This effort originates a specific strategy to the entire language as well as allows reuse of the theories and tools of the homogeneous base language [14].

This is exactly what the strategy presented in [21] does for CSP_Z . Since CSP_Z gives a failures-divergences semantics to the Z part, it is proposed a way of translating the Z part into a CSP process and capturing the semantics of the whole using some CSP operator. This also allows reusing the FDR tool [10], the standard model checker of CSP.

This CSP representation for a CSP_Z process is presented, in a simplified way, in the following definition. This definition also points out another interesting feature of the translation: the Z part assumes a CSP normal form suitable for our kind of data abstraction analysis.

Definition 1 (Normal Form of CSP_Z Processes). Let P be a CSP_Z specification. Let P_{CSP} and P_Z be the CSP and Z parts of P , respectively, as CSP processes. Then, the CSP representation of P , P_{CSP_Z} , is defined as

$$P_{CSP_Z} = (P_Z^{State} \parallel P_{CSP}) \setminus L,$$

where I is the synchronisation interface (all declared channels) and L is the set of local/hidden channels ($L \subseteq I$).

The Z part is given by a parameterised process P_Z^{State} of the form

$$P_Z^{State} = \left(\begin{array}{l} pre\ com\_chan_1 \ \& \ chan_1 \rightarrow P_Z^{com\_chan_1(State)} \\ \square pre\ com\_chan_2 \ \& \ chan_2 \rightarrow P_Z^{com\_chan_2(State)} \\ \vdots \\ \square pre\ com\_chan_n \ \& \ chan_n \rightarrow P_Z^{com\_chan_n(State)} \\ \square terminate \rightarrow SKIP \end{array} \right)$$

where $State$ is a tuple representing the system state, schemas are represented by generic functions and preconditions become boolean functions. $\diamond$

The synchronisation interface ($I = \{chan_1, \dots, chan_n, terminate\}$) contains all channels declared in the original specification and the new event *terminate*, which is used to synchronise the CSP and Z parts when the CSP part can terminate successfully. Local channels are used only in the synchronisation between P_{CSP} and P_Z^{State} . Therefore, they are hidden in the top level process (P_{CSP_Z}).

The process which captures the Z part (P_Z^{State}) can be seen as a passive process offering all channels of the interface when the respective schema precondition is valid ($pre\ com\_chan_i$). After engaging into some event ($chan_i$), P_Z changes the state according to the corresponding schema (com_chan_i).

It is worth pointing out that CSP_Z specifications themselves can be composed; such a composition can have synchronisation on specific channels. In such a context, the abstraction produced by our approach (for one CSP_Z process) does not affect the behaviour of the whole composition. The reason is that processes can only interact with others through channels. They do not exchange data. Thus, as our approach preserves the behaviour of a process, its abstract version continues interacting with other processes in the same way as the concrete version.

The rest of this work considers that a CSP_Z process has already been translated according to Definition 1. See [21,6] for more details of this translation.

4 Data Abstraction

One of the most powerful tools to allow model checking to be performed on infinite state systems is abstraction [12]. It consists of replacing an infinite state system for a finite state system while retaining most of its original properties [32].

Now we consider how an infinite state CSP_Z process can be abstracted. Such an abstraction is defined in terms of the theory of data independence [31,18] for the behavioural part and abstract interpretation [2] for the data part.

4.1 Data Independence

A system P is data independent (with respect to a data type X), if it does not perform any operation involving values of X ; it can only input such values, store them, and compare them for equality. In that case, the behaviour of P is preserved if any concrete data type (which admits equality) is substituted for X .

The work reported in [18] has used data independence to the specific context of analysing refinement relations between infinite state CSP processes. This work defines levels of data independence which allows one to check more or less detailed properties of systems. Such levels of data independence are presented in the following definition. Thus, the more restrictions a system satisfies the more properties can be checked.

Definition 2 (Data Independence). *P is data independent in a type X iff:*

1. *Constants do not appear in P , only variables appear, and*
2. *If operations are used then they must be polymorphic, or*
3. *If comparisons are done then only equality tests can be used, or*
4. *If used, complex functions and predicates must originate from 2 and 3, or*
5. *If replicated operators are used then only nondeterministic choices over X may appear in P .* $\diamond$

In particular, these levels are characterised by cardinality constraints (threshold collections) imposed on the data independent types present in a process. For example, suppose that $P \sqsubseteq_M Q$ (where M stands for some CSP semantical model) has to be checked, such that P and Q are infinite state and data independent with respect to an infinite data type X . After analysing the syntax of P and Q , the work [18] classifies the data independence level in terms of a cardinality constraint ($\#X \geq N$). Hence this refinement relation does not have to be checked for all values of X , but simply for a sufficient subset of it. For instance, in [18] the data type X is replaced with the subset of the natural numbers $\{0, 1, \dots, N-1\}$ and the refinement is checked using the FDR tool [10].

Based on this idea, definitions 3 and 4 have been proposed for isolating the behavioural part from the data one. These allow a compositional CSP_Z abstraction by only investigating the Z part.

Definition 3 (Trivial Data Independence). *A trivially data independent CSP process is a data independent process with no equality tests and polymorphic operations, ensuring $\#X \geq 1$ for all data type X independent in P .* $\diamond$

Definition 4 (Partial Data Independence). *A CSP_Z specification is partially data independent if its CSP part is trivially data independent.* $\diamond$

Although our approach deals with two parts of a CSP_Z specification separately, data independence should be considered in the CSP part because it cannot be affected when abstracting data types (confined to the Z part).

4.2 CSP_Z Data Abstraction

A data abstraction consists basically on replacing concrete data and related operations (possibly infinite) for abstract versions of corresponding data (possibly finite) and operations while still preserving most of the original properties of the abstracted system. It is based on the theory of abstract interpretation [2].

The work reported in [28] has investigated such a data abstraction for CSP_{OZ}, an object-oriented version of CSP_Z. But this work did not state what kinds of restrictions the CSP part must have. Without such restrictions a valid abstraction for the Z part cannot be valid for the whole CSP_{OZ} process. Furthermore, the abstractions are not achieved mechanically.

Further progress has been achieved in [20,22] where the CSP part is required to be trivially data independent (Definition 3), or the entire CSP_Z process to be partially data independent (Definition 4), in order to abstract a CSP_Z process by abstracting only its Z part correctly. Another contribution of this effort was the first mechanised approach for a data abstraction of an integrated language.

Unfortunately, some CSP_Z processes cannot be abstracted by the approach presented in [20,22]. The reason is simple: it only abstracts the Z part. Thus, a more powerful approach is required in order to consider a wider class of CSP_Z infinite state processes. This is the purpose of the next section.

Recall from Definition 1 that a Z schema is now represented as a function

$$com\_c : D \rightarrow \mathbb{P}(D)$$

where D is a tuple $D = D_1 \times \dots \times D_n$ representing the state space, assuming that there are n state components with types $D_1, \dots, D_n$, respectively.

The abstract version of this function is another function

$$com\_c^A : D^A \rightarrow \mathbb{P} D^A$$

where D^A is the abstract state domain (i.e. $D^A = D_1^A \times \dots \times D_n^A$) corresponding to D in terms of an abstraction function h

$$h(d) = (h_1(d_1), \dots, h_n(d_n))$$

where each h_i has type $D_i \rightarrow D_i^A$.

Furthermore, according to the abstract interpretation theory [2], the construction of com_c^A is compositional in the sense that the abstract version is obtained from abstracting its inner operations. For example, let s, s_1, s_2 be variables whose type is the powerset of some type, and com_c be a concrete operation which has the inner operation $s = s_1 \cup s_2$ ($\cup$ is the usual union). Thus, the abstract function com_c^A is given by $s^A = s_1^A \cup^A s_2^A$. Note that in the abstract version we have another operation for union ($\cup^A$), which can be a slight variant of the original one as was argued in the present work.

An operation can have many abstract versions. In this work we are interested in abstractions which preserve the behaviour of the whole specification. We focus on optimal abstraction because it represents the system more faithfully. The following definition states the optimal abstraction of an operation [2].

Definition 5 (Optimal Abstraction). *An operation com_c^A is the optimal abstraction of com_c , according to an abstraction function h , iff*

$$\forall d : D \bullet (com\_c^A \circ h)(d) = (h \circ com\_c)(d) \quad \diamond$$

As presented in [20,22], the abstract data and operations can be found mechanically. Broadly, an algorithm identifies a class of values (possibly an infinite partition) of the original type with a single value that represents the entire class (partition). A limitation of this approach is that the abstraction is inferred by considering only the Z part. Therefore, the CSP part is not considered as a controller process, and some situations, specific of the CSP part, are not captured.

Next section presents an algorithm which mechanises the data abstraction approach and also considers the influence of the CSP part. This allows one to find data abstractions faster and more specific to the process being analysed.

5 Algorithm

The basic idea behind CSP_Z data abstraction is avoiding infinite expansions by finding out stable behaviour of the process being analysed. An infinite stable (or periodic) behaviour is one that can be represented by a finite LTS. For example, the process $P = a \rightarrow P$ is stable because it has the infinite trace $\langle a, a, \dots \rangle$ and can be represented by the LTS of Figure 2 (a finite LTS).

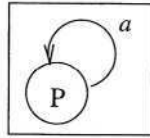


Fig. 2. A Simple Stable Process

Stability can be determined only for the Z part by observing the repetition of a sequence of properties, where a property stands for a conjunction of enabled and disabled schemes [20,22]. For example, for a CSP_Z process with operations com_a and com_b , the property $pre\ com\_a \wedge \neg pre\ com\_b$ means that the event a can occur and b cannot.

Note that the above property does not consider any information about the CSP part. In our approach, we extend this property by adopting the conjunction of the acceptances (and refusals complementarily) of the whole process.

From Example 1, we can state that the property before and after performing the trace $\langle tick, tack \rangle$ is given by

$$tick \in initials(P_{CSP_Z}) \wedge tack \notin initials(P_{CSP_Z})$$

where $initials(P_{CSP_Z}) = \{ev \mid ev \in initials(P_{CSP}) \wedge pre\ com\_ev\}$. It is worth noting that $initials(P_{CSP_Z})$ takes into account the CSP ($initials(P_{CSP})$) and Z ($pre\ com\_ev$) parts simultaneously.

One of our goals in this paper is to show how to determine whether a CSP_Z property repeats infinitely. We do this by, first, looking for cycles in the CSP part; for instance, the CSP part of the process *Clock* of Example 1 performs $\langle tick, tack \rangle$. And after finding one of them, determining if the Z part repeats the same property as well; for Example 1, this leads us to prove a theorem (Section 5.1) involving the composition $com\_tick \circ com\_tack$.

5.1 Determining Behavioural Stability

One step of the algorithm consists of gradually expanding a CSP_Z process based on its operational semantics [9]. But prior to perform this task, the algorithm checks whether the successor state causes the repetition of some property. If so, the algorithm investigates the stability of the corresponding behaviour.

Recall the process *Clock* from Section 2 (see Figure 6). It repeats the property

$$tick \in initials(P_{CSP_Z}) \wedge tack \notin initials(P_{CSP_Z})$$

before and after performing the trace $\langle tick, tack \rangle$. So, instead of a further expansion, the algorithm checks whether the CSP and the Z parts repeat such a property infinitely.

For the CSP part it means that the behaviour of *main* (in terms of trace), when the probable stable point has been reached, must be equivalent to an infinite and stable process ($Prop_{CSP}$), that is

$$main \equiv_{\mathcal{T}} Prop_{CSP}$$

where $Prop_{CSP} = tick \rightarrow tack \rightarrow Prop_{CSP}$.

As a consequence of the definition of $\equiv_{\mathcal{T}}$, the above equivalence can be performed as the refinement check $(main \sqsubseteq_{\mathcal{T}} Prop_{CSP}) \wedge (Prop_{CSP} \sqsubseteq_{\mathcal{T}} main)$. The first part informs that $Prop_{CSP}$ does not produce any trace different from *main*, when it reaches the stable point, and the second part, concerning the infiniteness of *main*, says that all traces produced by *main*, when it reaches the stable point, are also produced by $Prop_{CSP}$.

For the Z part we have to determine if the composition $com\_tick \circ com\_tack$ is always possible, proving the theorem

$$\forall State; State' \mid (pre\ comp \Rightarrow comp) \bullet (pre\ comp)'$$

where *comp* captures the desired schema composition.

It is worth pointing out that [20,22] employ a slightly different predicate

$$\forall State; State' \mid pre\ comp \bullet comp \Rightarrow (pre\ comp)'$$

We have observed that there is a subtle situation which is not captured by this latter predicate. If one schema of *comp* is disabled, then *pre comp* is not valid and, therefore, no change occurs (that is, $(pre\ comp)'$ is not valid either). Rewriting the predicate to a simpler form, and using boolean values we obtain $false \Rightarrow (false \Rightarrow false)$, which produces *true* when *false* is expected (the system is not stable).

5.2 Description

Now we present our algorithm for CSP_Z data abstraction which takes into account the CSP and Z parts, simultaneously, as briefly described in the previous section. It is worth pointing out that the algorithmic parts are intentionally presented in an informal way to ease understanding (see [5] for formal details).

The execution model of our algorithm is based on a structure which contains information about state, LTS of the CSP part, performed trace, property and a sequence of nodes (path). Such a compound data is stored inside a node or a state of an LTS (see the left-hand side of Figure 3). Beyond states, an LTS also has transitions which are labelled by the CSP event performed as well as the corresponding Z schema operation (see the right-hand side of Figure 3).

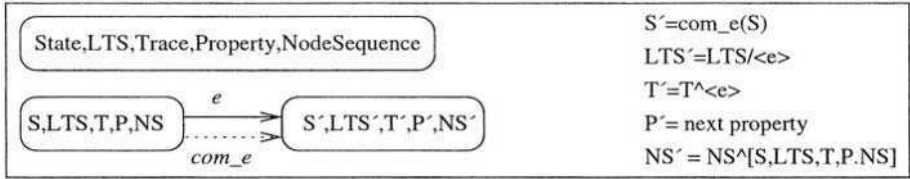


Fig. 3. Execution model

To keep such an information, our algorithm uses specific data structures, which are detailed in Table 1. Figure 4 presents the algorithm itself.

Table 1. Structures and variables of the algorithm

Node	Node=(State,LTS,Trace,Property,NodeSequence)
CurrentNode	The current node to be processed.
CurrentLTS	The LTS representing the CSP part of the current node.
CurrentTrace	The trace performed by the process up to CurrentNode.
CurrentProperty	The property of the current node.
CurNodeSequence	The sequence of nodes used to reach current node.
VisitedNodes	All nodes created by the algorithm.
CurrentStage	A set of nodes to be processed.
NextStage	A set of nodes which are children of the current nodes.
first, second, third, fourth	Return a node's component. That is, first(Node) returns State, second(Node) returns LTS, and so on.
extractProperty	Gives a node's property based on a specific state and LTS
extractIndex	Gives the index of a node on a sequence of nodes
extractTrace	Gives the subtrace of a trace from a specific index

The construction of the first node is presented from line 1 to line 7. The initial state is produced by executing the schema *Init*, and the initial LTS of the CSP

part is built by the function `buildInitialLTS()`. Afterwards, the algorithm performs a breadth first search, where the children nodes are analysed after all parent nodes have been analysed.

```

1. CurrentState := Init
2. CurrentTrace := {}
3. CurrentLTS := buildInitialLTS()
4. CurrentProperty := extractProperty(CurrentState, CurrentLTS)
5. CurNodeSequence := {}
6. CurrentNode := (CurrentState, CurrentLTS, CurrentTrace,
7.               CurrentProperty, CurNodeSequence)
8. VisitedNodes := {CurrentNode}
9. CurrentStage := {CurrentNode}
10. NextStage := {}

11. while (CurrentStage has more nodes)
12.   ∀ node ∈ CurrentStage
13.     CurrentNode := node
14.     ∀ com_e enabled
15.       if e ∈ initials(CurrentLTS)
16.         if not checkStable(com_e, CurrentNode)
17.           NewState := com_e(CurrentState)
18.           NewLTS := CurrentLTS / (e)
19.           NewTrace := CurrentTrace ^ (e)
20.           NewProperty := extractProperty(NewState, NewLTS)
21.           NewNodeSequence := CurNodeSequence ^ (CurrentNode)
22.           NewNode := (NewState, NewLTS, NewTrace,
23.                     NewProperty, NewNodeSequence)
24.           if NewNode ∉ VisitedNodes
25.             VisitedNodes := VisitedNodes ∪ {NewNode}
26.             NextStage := NextStage ∪ {NewNode}
27.         fi
28.       fi
29.     fi
30.   ∀-end
31. ∀-end
32. CurrentStage := NextStage
33. NextStage := {}
34. while-end

```

Fig. 4. CSP_Z Abstraction Algorithm

When processing a node, the algorithm takes all enabled schemas at that node (line 14) and asks whether the CSP part accepts one such a corresponding event (line 15). If so, it also checks the stability of the system (line 16).

The function `checkStable`, which takes into account the CSP and the Z parts, returns *true* iff the current trace constitutes an infinite stable behaviour

(see Figure 5). The function `checkCycles` determines if the CSP part has or not a cycle performing a given trace, after the stable point has been reached. This function has been implemented in our tool instead of checking the equivalence $main \equiv_{\tau} Prop_{CSP}$ by refinement, as explained in Section 5.1.

If the CSP part is not stable, the next node has to be exploited (lines 17–23).

```

checkStable(com_op, Node){
  isCspzCycle := false
  State' := com_op(first(Node))
  LTS' := second(Node)/op
  Property' := extractProperty(State', LTS')
  indexofRepetition := extractIndex(Property', fifth(Node))
  stableTrace := extractTrace(third(Node), indexofRepetition)
  if (index != 0)
    if (checkCycles(stableTrace))
      comp := the composition corresponding to stableTrace
      z-theorem :=  $\forall State; State' \mid (pre\ comp \Rightarrow comp) \bullet pre\ comp'$ 
      isCspzCycle := (ask z-theorem to a theorem prover)
    fi
  fi
  return isCspzCycle
}

```

Fig. 5. Stability checking function

The following theorem states that our algorithm finds a optimal abstraction of a CSP_Z process, if it exists. The proof can be found in [5].

Theorem 1 (Soundness). *Let P be an infinite state CSP_Z process. A finite optimal abstraction for P is achieved if our algorithm terminates successfully.*

◇

5.3 Building the Abstraction

Assuming that our algorithm has terminated successfully, we start the construction of the abstract process. The steps are explained by using Example 1 of Section 2.

1. Process *Clock* is expanded until the state $c = 1$ is achieved (this is illustrated in Figure 6(a)). Note that both parts are expanded simultaneously where the behavioural part guides the expansion task;
2. Before actually generating the node where $c = 2$, the algorithm looks ahead (at that state) and detects the repetition of the property

$$tick \in \text{initials}(P_{CSP_Z}) \wedge tack \notin \text{initials}(P_{CSP_Z})$$

Then, it calls the function `checkStable` which checks the following questions

$$main \equiv_{\tau} Prop_{CSP}$$

where $Prop_{CSP} = tick \rightarrow tack \rightarrow Prop_{CSP}$, and

$$\forall State; State' \mid (pre \text{ comp} \Rightarrow comp) \bullet (pre \text{ comp})'$$

where $comp \hat{=} com\_tick \text{ } \& \text{ } com\_tack$;

3. As both questions are valid the algorithm concludes that the current expansion is stable and thus no further expansion is needed. Instead, a τ -transition is generated from the current state $c = 1$ to the state where the property has firstly occurred. This is equivalent to replace the state $c = 2$ for state $c = 0$, fact proved in the previous step, as illustrated in Figure 6(b);
4. Follow another branch of the CSP_Z process behaviour. As Example 1 does not have further branches to investigate the algorithm starts the abstraction;
5. The abstraction of the state variables is obtained directly from the finite LTS (see Figure 6(b)). That is, instead of the original definition $c \in \mathbb{N}$ we have $c^A \in \{0, 1\}$, or $(\mathbb{N}^A \hat{=} \mathcal{A} = \{0, 1\})$;
6. The abstract state space is originated simply by using the abstract types created in the previous step, that is, $State^A \hat{=} [c^A : \mathcal{A}]$;
7. The abstraction of the initialisation is obtained by replacing the effect of *Init* with an assignment to the state variable, such that its value is extracted from the initial node of the LTS, that is, $Init^A \hat{=} [(State^A)'] \mid (c^A)' = 0]$.
8. The abstraction of the operations is slightly different. While the precondition is always preserved the postcondition can change. If a schema belongs to a cycle, its postcondition is rewritten to produce the value taken from the LTS (see Figure 6.b); otherwise, it is preserved. For example, *com_tick* and *com_tack* belong to the cycle $\langle tick, tack \rangle$. Therefore, their abstract postconditions become $c' = 1$ and $c' = 0$, respectively.

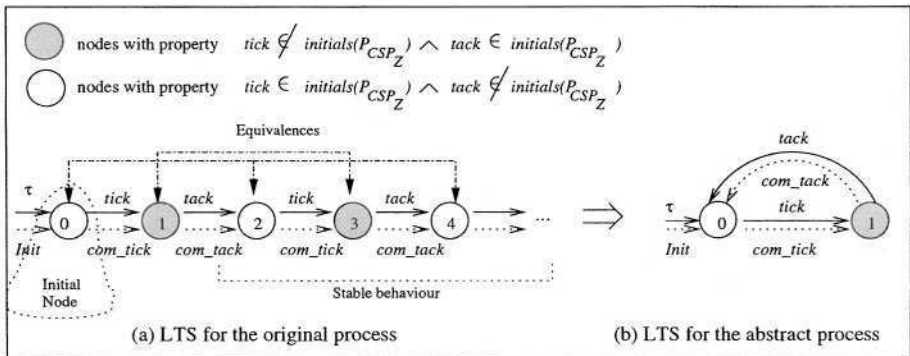


Fig. 6. The abstraction of *Clock*

Finally, we have the process $Clock^A$ (the abstract version of $Clock$).

spec $Clock^A$

chan $tick, tack$

$main = tick \rightarrow tack \rightarrow main$

$\mathcal{A} = \{0, 1\}$

$State^A \triangleq [c^A : \mathcal{A}]$

$com\_tick \triangleq [\Delta State^A \mid (c^A)' = 1]$

$Init^A \triangleq [(State^A)' \mid (c^A)' = 0]$

$com\_tack \triangleq [\Delta State^A \mid (c^A)' = 0]$

end_spec $Clock^A$

where the superscript A , used inside the scope of $Clock^A$, is just to emphasise the abstract version of the corresponding elements.

It is worth noting that we do not calculate any abstraction function. But from the above specification and the original one we can infer it:

$$h(x) = \begin{cases} 0, & \text{if } x \bmod 2 = 0 \\ 1, & \text{if } x \bmod 2 = 1 \end{cases}$$

The above function is a mapping whose range gives the abstract domain, and its application to the schemas gives the following abstract versions:

$State \triangleq [c : \{0, 1\}]$

$com\_tick \triangleq [\Delta State \mid c' = h(h(c) + 1)]$

$Init \triangleq [State' \mid c' = 0]$

$com\_tack \triangleq [\Delta State \mid c' = h(h(c) + 1)]$

5.4 Performance and Termination

Besides finding an equivalent data abstraction as previous work [22], our algorithm is far more efficient in a large number of situations. In this section we explain the reason of this performance improvement.

From the previous sections, it is clear that our algorithm is based on the exploration of the possible traces a CSP_Z can exhibit. Thus, we justify our improvement in terms of the amount of traces needed in order to abstract a CSP_Z process by considering its CSP and Z parts simultaneously, instead of simply its Z part [20,22].

A very simple notion of refinement for CSP is traces refinement. A process Q trace refines ($\sqsubseteq_{\mathcal{T}}$) a process P iff Q exhibits less traces or the same as P . Such a notion is stated formally in Definition 6 below (see [13,23] for details).

Definition 6 (Traces Refinement). *Let P and Q be two CSP processes. Then,*

$$P \sqsubseteq_{\mathcal{T}} Q \Leftrightarrow \text{traces}(P) \supseteq \text{traces}(Q).$$

◇

Additionally, it is worth pointing out that a fully synchronised parallel combination, where all events of both participants must synchronise, originates less traces or the same as of one of its constituents. This is stated formally in Lemma 1 (see [5] for the proof). Therefore, it is reasonable to claim that a CSP_Z process

originates at most the same traces as its Z part. As a consequence, our algorithm, which considers the CSP and Z parts together, is faster than one that takes only the Z part [20,22].

Lemma 1 ($(P \parallel_I Q)$ **trace refines** P **or** Q). *Let P and Q be two CSP processes. Then,*

$$P \sqsubseteq_{\mathcal{T}} P \parallel_I Q, \text{ or } Q \sqsubseteq_{\mathcal{T}} P \parallel_I Q$$

where $I = \alpha P = \alpha Q$. ◇

Note that the above lemma also holds when the CSP part stops, terminates or diverges. These specific situations are captured by CSP laws of parallelism (see [13,23] for details).

Example 2 illustrates a typical situation of the above discussion. It presents a hypothetical CSP_Z process which deadlocks after performing the trace $\langle a, a, a \rangle$ (observe the precondition of com_b). But if one explores only its Z part a large number of states is needed ($\approx 10^6$).

Example 2. A hypothetical CSP_Z process.

spec S

channel a, b

main $= a \rightarrow a \rightarrow a \rightarrow b \rightarrow \text{main}$

State $\hat{=} [n : \mathbb{N}]$

Init $\hat{=} [State' \mid n' = 0]$

com_a $\hat{=} [\Delta State \mid n < 10^6 \wedge n' = n + 1]$

com_b $\hat{=} [\Delta State \mid n \geq 10^6 \wedge n' = n + 1]$

end_spec S

Figure 7 shows the expansions of the above process according to the previous discussion. By considering only the Z part, one has to execute $10^6 + 1$ steps before finding out a stable point. For the CSP part however, we need only 4 expansions since after performing $\langle a, a, a \rangle$, the CSP part only accepts the event b whereas the Z part only accepts the event a . Both alternatives allow us to find finite abstractions for this example, but the compound approach is far less expensive than the modular approach.

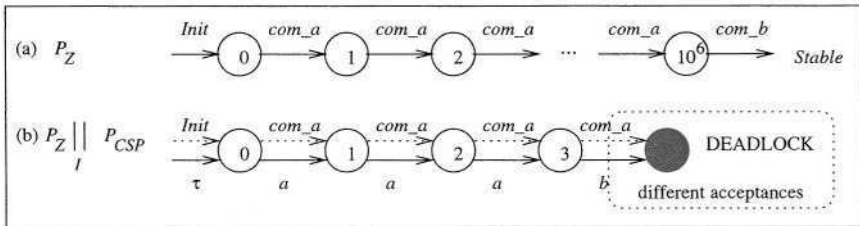


Fig. 7. Expansions of P_Z and $P_Z \parallel_I P_{CSP}$

It is important to single out that besides the impact on performance, the algorithm proposed here also finds an abstraction at least as often as the one presented in [21], but possibly more often. This is illustrated in Example 3, where the filtering originated by the CSP part allows the successful termination of our algorithm whereas the algorithm reported in [20,22] does not terminate.

Example 3. A terminating CSP_Z process with an unstable Z part.

spec *ND*

chan *a, b, c*

main = $a \rightarrow c \rightarrow a \rightarrow b \rightarrow STOP \sqcap c \rightarrow a \rightarrow c \rightarrow b \rightarrow SKIP$

State $\hat{=}$ $[x : \mathbb{N}]$

Init $\hat{=}$ $[State' \mid x' = 1]$

com_a $\hat{=}$ $[\Delta State \mid x \leq 5 \wedge x' = x + 1]$

com_b $\hat{=}$ $[\Delta State \mid x \geq 5 \wedge x' = x - 3]$

com_c $\hat{=}$ $[\Delta State \mid (x < 4 \vee x \geq 6) \wedge x' = 2 \times x]$

end_spec *ND*

This process can only perform the traces $\langle a, c, a, b \rangle$ or $\langle c, a, c, b \rangle$. So, our algorithm terminates successfully with a simple data abstraction. However, according to Figure 8 the Z part of this process is unstable (the Z stability theorem cannot be satisfied) and thus the algorithm reported in [20,22] does not terminate.

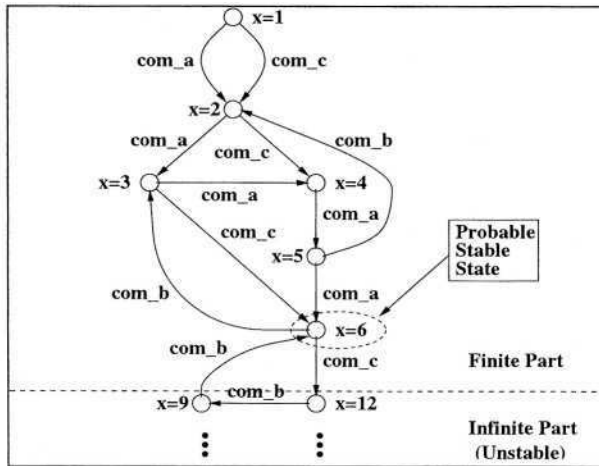


Fig. 8. LTS of an Unstable process

In general, this is also a consequence of Lemma 1. Therefore, we deal with a wider class of problem than the previous approach. Obviously, if the CSP part cannot filter such an instability then our algorithm diverges as well.

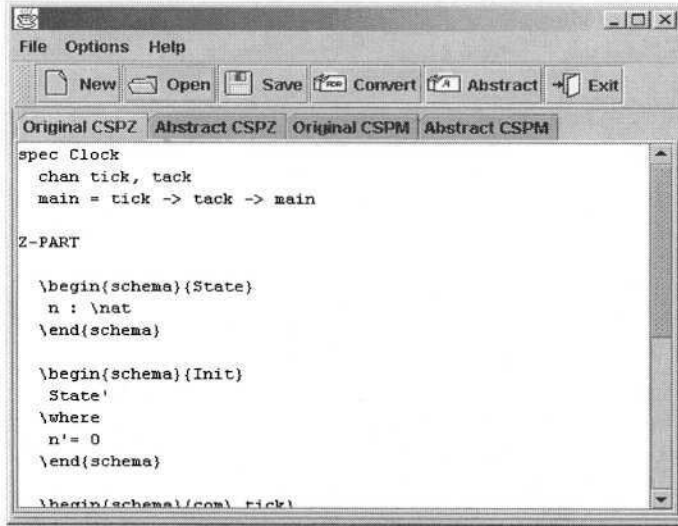


Fig. 9. Main screen of the tool

5.5 Tool Support

To provide tool support for CSP_Z data abstraction, we have developed a tool in Java¹ [15], which implements the algorithm of Section 5.2. The examples presented here and elsewhere [5] have been automatically abstracted using this tool conjointly with the Z-Eves [24] theorem prover. Figure 9 presents the front-end of the tool.

The functionality of the tool comprehends four main modules (see Figure 10): the *Parser* that reads a (CSP_Z specification) file and generates a syntactical tree, the *Translator* that converts a CSP_Z process into an equivalent CSP_M ² process [21], the *Data Independence* that determines whether the CSP part is partially data independent (see Definition 4) and the *Data Abstraction* that specifically implements the algorithm of Section 5.2.

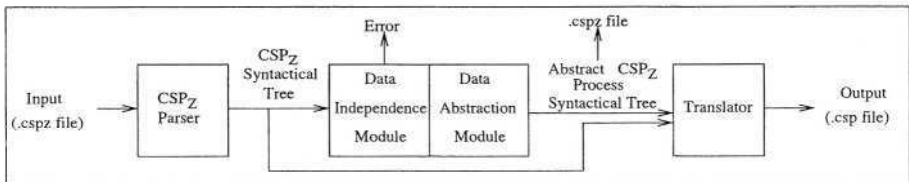


Fig. 10. The modules of the tool

¹ It is available for download at <http://www.cin.ufpe.br/~acf>.

² CSP_M is the machine-readable version of CSP used by the FDR tool.

6 Conclusions

In this paper we have addressed the problem of model checking infinite state systems. Although specifically related to the specification language CSP_Z , we have shown that our strategy in general consists in avoiding the expansion of a process indefinitely by looking for stable behaviours.

Instead of trying to build an infinite LTS, a task which indeed does not terminate, we examine each new generated state to see whether it is repeating a previous property (behavioural and data aspects simultaneously). In the case of a repetition we check further to determine if it is infinite. Concluding that it is indeed infinite, by a refinement check for the CSP part and a proof of a theorem for the Z part, we can stop expanding such a branch and add an invisible transition from the current state to the state starting this repetition.

Our contribution is presented in the form of an algorithm in Section 5.2 (Figure 4), where the function `checkStable` (Figure 5) is responsible for determining whether a property repetition is infinite or not. It is worth noting that, since `checkStable` takes into account a proof of a theorem, our algorithm is indeed semi-automatic. Furthermore, the algorithm is sound (Theorem 1) in the sense that if it terminates successfully then the process analysed can be data abstracted and such a data abstraction is optimal [2].

Comparing our proposal with that present in [20,22], we can benefit from a superior performance, more specific data abstractions, and dealing with a wider class of infinite state CSP_Z processes. Performance happens naturally by also considering the behavioural part of a CSP_Z process; a simple property of the CSP parallel operator, presented by Lemma 1, justifies this gain. More specific data abstractions comes directly from examining the entire CSP_Z process instead of only its data part. Finally, our proposal can abstract more CSP_Z processes because the behavioural part filters the data part (Lemma 1). That is, for those CSP_Z processes which do not have a finite (abstract) LTS corresponding to its data part, its filtered version (considering the CSP part) sometimes has (as illustrated in Example 3).

Concerning the proposals presented in [26,30], the distinguishing feature of our approach is that instead of using boolean abstractions (replacing predicates and expressions with boolean variables) we replace infinite types with corresponding subtypes.

A further contribution of our work is a robust prototype written in Java [15] (Section 5.5) which implements the algorithm of Section 5.2. The prototype offers facilities for translating a CSP_Z process into a CSP one, according to [21], before or after abstracting the process. The implementation of the function `checkStable` needs the aid of a theorem prover. The prototype was designed in such a way that working with different theorem provers is easy; it is only necessary a configuration file. The examples presented in this paper have been developed with the help of this prototype using the Z-Eves theorem prover [24].

As future work we intend to deal with unstable processes where a finite LTS representation is not trivially achievable by a mechanised approach. This was briefly illustrated in Example 3 where the Z part could not be abstracted. Thus

allowing a more flexible CSP part, instead of a finite one as in Example 3, can avoid the mechanised abstraction even using our approach.

Another research direction is to consider certain common data dependencies originated from the behavioural part, such as data communications restrictions. We also plan to instantiate our proposal to decidable theories where the algorithm can be fully automatised.

References

1. Cleaveland, R. and Riely, J. Testing-based abstractions for value-passing systems. In J. P. B. Jonsson, editor, *CONCUR'94*, volume 836, pages 417–432. Springer-Verlag, Berlin, 1994.
2. Cousot, P. and Cousot, R. Abstract interpretation frameworks. *Journal of Logic and Computation*, **2(4)**:511–547, 1992.
3. Duke, R., Rose, G. and Smith, G. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*. **17**:511–533, 1995.
4. Ehrig, H., Fey, W. and Hansen, H. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-01, Technische Universität Berlin, 1983.
5. Farias, A. Efficient and Mechanised Analysis of Infinite CSP_Z Processes: strategy and tool support. M.Sc. dissertation, 2003.
6. Farias, A., Mota, A. and Sampaio, A. From CSP_Z to CSP_M : a Transformational Java Tool. In *Proceedings of IV Workshop on Formal Methods*, Computing Brazilian Society (SBC), 2001, pp. 1–10.
7. Fischer, C. Combining CSP and Z. Technical Report, University of Oldenburg, 1996.
8. Fischer, C. *Combination and Implementation of Processes and Data: from CSP-OZ to Java*. PhD thesis, Fachbereich Informatik Universität Oldenburg, 2000.
9. Fischer, C. CSP-OZ: a combination of object-Z and CSP. *2nd IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'97)*, Chapman & Hall, London, 1997.
10. Formal Systems (Europe). FDR2 User Manual, 1997.
11. Galloway, A. *Integrated formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems*. PhD thesis, University of Teesside, School of Computing and Mathematics, 1996.
12. Grumberg, O., Clarke, E. and Peled, D. *Model Checking*. The MIT Press, Cambridge, MA, 1999.
13. Hoare, C. A. R. *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ, 1985.
14. Hoare, C.A.R and Jifeng, H. *Unifying Theories of Programming*. Prentice-Hall, 1998.
15. Horstman, C. and Cornell, G. *Core Java 2*. Sun Microsystems Press, volumes I and II, 2000.
16. ISO. Information technology - Programming languages, their environments and system software interfaces - *Vienna Development Method - Specification Language* - Part 1: Base language. International Standard ISO/IEC 13817-1, December 1996.

17. ISO. Information Processing Systems - Open Systems Interconnection - *LOTOS - A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807, International Organisation for Standardisation, Geneva, Switzerland, 1989.
18. Lazić, R. *A Semantic Study of Data Independence with Applications to Model Checking*. PhD thesis, Oxford University Computing Laboratory, 1999.
19. Milner, R. In *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol. 92, Springer-Verlag, Berlin, 1980.
20. Mota, A. *Model Checking CSP_Z: Techniques to Overcome State Explosion*. PhD thesis. Federal University of Pernambuco, Brazil, 2002.
21. Mota, A. and Sampaio, A. Model-Checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming*, Elsevier, Netherlands. (40)1:59–96, 2001.
22. Mota, A., Sampaio, A. and Borba, P. *Mechanical Abstraction of CSP_Z Processes*. FME 2002, LNCS 2391, pp. 163–183.
23. Roscoe, A. W. *The Theory and Practice of Concurrency*. Oxford University, 1998.
24. Saaltink, M. Z-Eves System. In ZUM'97: the Formal Specification Notation, volume 1212, LNCS, Springer, 1992.
25. Spivey, M. *The Z Notation: A Reference Manual, 2nd Edition*. Prentice Hall International, Englewood Cliffs, NJ, 1992.
26. Stahl, K., Baukus, K., Lakhneich, Y. and Steffen, M. *Divide, Abstract and Model Check*. SPIN, pp. 57–76, 1999.
27. Toetenal, W. *Model-Oriented Specification of Communicating Agents*. PhD thesis, Faculty of Mathematics and Informatics, 1992.
28. Wehrheim, H. Data Abstraction for CSP-OZ. *FM'99 World Congress on Formal Methods*, Lecture Notes in Computer Science, vol. 1709, Springer, Berlin, 1999.
29. Woodcock, J. and Cavalcanti, A. *The Semantics of Circus*. In Didier Ber, Jonathan P. Bowen, Martin C. Henson and Ken Robinson, editors, ZB 2002: Formal Specification and Development in Z and B, LNCS, 2272:184–203. Springer-Verlag, 2002.
30. Namjoshi, K. S. and Kurshan, R. P. *Syntactic Program Transformations for Automatic Abstraction*. Computer Aided Verification, pp 435–449, 2000.
31. Wolper, P. *Expressing Interesting Properties of Programs in Propositional Temporal Logic*, pp. 184–192, Proc. 13th ACM Symp. on Principles of Programming Languages, 1986.
32. Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A. and Bensalem, S. *Property Preserving Abstractions for the Verification of Concurrent Systems*, pp. 11–44, Formal Methods in System Design, volume 6, number 1, 1995.

State/Event-Based Software Model Checking*

Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina, and
Nishant Sinha

Computer Science Department, Carnegie Mellon University
5000 Forbes Ave., Pittsburgh PA 15213, USA

Abstract. We present a framework for model checking concurrent software systems which incorporates both states and events. Contrary to other state/event approaches, our work also integrates two powerful verification techniques, counterexample-guided abstraction refinement and compositional reasoning. Our specification language is a state/event extension of linear temporal logic, and allows us to express many properties of software in a concise and intuitive manner. We show how standard automata-theoretic LTL model checking algorithms can be ported to our framework at no extra cost, enabling us to directly benefit from the large body of research on efficient LTL verification.

We have implemented this work within our concurrent C model checker, MAGIC, and checked a number of properties of OpenSSL-0.9.6c (an open-source implementation of the SSL protocol) and Micro-C OS version 2 (a real-time operating system for embedded applications). Our experiments show that this new approach not only eases the writing of specifications, but also yields important gains both in space and in time during verification. In certain cases, we even encountered specifications that could not be verified using traditional pure event-based or state-based approaches, but became tractable within our state/event framework. We report a bug in the source code of Micro-C OS version 2, which was found during our experiments.

1 Introduction

Control systems ranging from smart cards to automated flight controllers are increasingly being incorporated within complex software systems. In many instances, errors in such systems can have dramatic consequences, hence the urgent need to be able to ensure and guarantee their correctness.

* This research was sponsored by the Semiconductor Research Corporation (SRC) under contract no. 99-TJ-684, the National Science Foundation (NSF) under grants no. CCR-9803774 and CCR-0121547, the Office of Naval Research (ONR) and the Naval Research Laboratory (NRL) under contract no. N00014-01-1-0796, the Army Research Office (ARO) under contract no. DAAD19-01-1-0485, and was conducted as part of the Predictable Assembly from Certifiable Components (PACC) project at the Software Engineering Institute (SEI). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of SRC, NSF, ONR, NRL, ARO, the U.S. Government or any other entity.

In this endeavor, the well-known methodology of *model checking* [CE81, CES86, QS81, CGP99] holds much promise. Although most of its early applications dealt with hardware and communication protocols, model checking is increasingly used to verify software systems [SLA, BR01, BMMR01, BLA, HJMS02, HJMQ03, CDH⁺00, PDV01, Sto02, MAG, CCG⁺03, COYC03]. Unfortunately, applying model checking to software is complicated by several factors, ranging from the difficulty to model computer programs—due to the complexity of programming languages as compared to hardware description languages—to difficulties in specifying meaningful properties of software using the usual temporal logical formalisms of model checking. A third reason is the perennial state space explosion problem, whereby the complexity of verifying an implementation against a specification becomes prohibitive.

The most common instantiations of model checking to date have focused on finite-state models and either branching-time (CTL [CE81]) or linear-time (LTL [LP85]) temporal logics. To apply model checking to software, it is necessary to specify (often complex) properties on the finite-state abstracted models of computer programs. The difficulties in doing so are even more pronounced when reasoning about *modular* software, such as concurrent or component-based sequential programs. Indeed, in modular programs, communication among modules proceeds via actions (or events), which can represent function calls, requests and acknowledgments, etc. Moreover, such communication is commonly data-dependent. Software behavioral claims, therefore, are often specifications defined over combinations of program actions and data valuations.

Existing modeling techniques usually represent finite-state machines as finite annotated directed graphs, using either *state-based* or *event-based* formalisms. Although both frameworks are interchangeable (an action can be encoded as a change in state variables, and likewise one can equip a state with different actions to reflect different values of its internal variables), converting from one representation to the other often leads to a significant enlargement of the state space. Moreover, neither approach on its own is practical when it comes to modular software, in which actions are often data-dependent: considerable domain expertise is then required to annotate the program and to specify proper claims.

This work, therefore, proposes a framework in which both state-based and action-based properties can be expressed, combined, and verified. The modeling framework consists of *labeled Kripke structures* (LKS), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions. The specification logic is a *state/event* derivative of LTL. This allows us to represent both software implementations and specifications directly without any program annotations or privileged insights into program execution. We further show that standard efficient LTL model checking algorithms can be applied, *at no extra cost in space or time*, to help reason about state/event-based systems. We have implemented our approach within the concurrent C verification tool MAGIC [MAG, CCG⁺03, COYC03], and report promising results in the examples which we have tackled.

The state/event-based formalism presented in this paper is suitable for both sequential and concurrent systems. One of the benefits of restricting ourselves to linear-time logic (as opposed to a more expressive logic such as CTL* or the modal μ -calculus) is the ability to invoke the MAGIC compositional abstraction refinement procedures developed for the efficient verification of concurrent software [COYC03]. These procedures are embedded within a counterexample-guided abstraction refinement framework (CEGAR for short) [CGJ⁺00], one of the core features of MAGIC. CEGAR lets us investigate the validity of a given specification through a sequence of increasingly refined abstractions of our system, until the property is either established or a real counterexample is found. Moreover, thanks to compositionality, the abstraction, counterexample validation, and refinement steps can all be carried out component-wise, thereby alleviating the need to build the full state space of the distributed system.

We illustrate our state/event paradigm with a current surge protector example, and conduct further experiments with the source code for OpenSSL-0.9.6c (an open-source implementation of the SSL protocol) and Micro-C OS version 2 (a real-time operating system for embedded applications). In the case of the latter, we discovered a bug, which it turns out was already known to the implementors of Micro-C OS. We contrast our approach with equivalent pure state-based and event-based alternatives, and show that the state/event methodology yields significant gains in human effort (ease of expressiveness), state space, and verification time, at no discernible cost.

The paper is organized as follows. In Section 2, we review and discuss related work. Section 3 defines our state/event implementation formalism, labeled Kripke structures. We also lay the basic definitions and results needed for the presentation of our compositional CEGAR verification algorithm. In Section 4, we present our state/event specification formalism, based on linear temporal logic. We review standard automata-theoretic model checking techniques, and show how these can be adapted to the verification task at hand. In Section 5, we illustrate these ideas by modeling a simple surge protector. We also contrast our approach with pure state-based and event-based alternatives, and show that both the resulting implementations and specifications are significantly more cumbersome. We then use MAGIC to check these specifications, and discover that the non-state/event formalisms incur important time and space penalties during verification.¹ Section 6 details our compositional CEGAR algorithm. In Section 7, we report on case studies in which we checked specifications on the source code for OpenSSL-0.9.6c and Micro-C OS version 2, which led us to the discovery of a bug in the latter. Finally, Section 8 summarizes the contributions of the paper and outlines several avenues for future work.

¹ In order to invoke MAGIC, we code the LKSs as simple C programs; the algorithm used by MAGIC implements the techniques described in the paper. Lack of space prevents us from discussing *predicate abstraction*, whereby MAGIC transforms a (potentially infinite-state) C program into a finite-state machine. We refer the reader to [CCG⁺03] for a detailed exposition of this point.

2 Related Work

Counterexample-guided abstraction refinement [CGJ⁺00,Kur94], or CEGAR, is an iterative procedure whereby spurious counterexamples to a specification are repeatedly eliminated through incremental refinements of a conservative abstraction of the system. CEGAR has been used, among others, in [NCOD97] (in non-automated form), and [BR01,PDV01,LBBO01,HJMS02,CCK⁺02,CGKS02,COYC03].

Compositionality, which features centrally in our work, is broadly concerned with the preservation of properties under substitution of components in concurrent systems. It has been extensively studied, among others, in process algebra (e.g., [Hoa85,Mil89,Ros97]), in temporal logic model checking [GL94], and in the form of assume-guarantee reasoning [McM97,HQR00,CGP03].

The combination of CEGAR and compositional reasoning is a relatively new approach. In [BLO98], a compositional framework for (non-automated) CEGAR over data-based abstractions is presented. This approach differs from ours in that communication takes place through shared variables (rather than blocking message-passing), and abstractions are refined by eliminating spurious transitions, rather than by splitting abstract states.

The idea of combining state-based and event-based formalisms is certainly not new. De Nicola and Vaandrager [NV95], for instance, introduce ‘doubly labeled transition systems’, which are very similar to our LKSs. From the specification point of view, our state/event version of LTL is also subsumed by the modal μ -calculus [Koz83,Pnu86,BS01], via a translation of LTL formulas into Büchi automata. The novelty of our approach, however, is the way in which we efficiently integrate an expressive state/event formalism with powerful verification techniques, namely CEGAR and compositional reasoning. We are able to achieve this precisely because we have adequately restricted the expressiveness of our framework. To our knowledge, our work is the first to combine these three features within a single setup.

Kindler and Vesper [KV98] propose a state/event-based temporal logic for Petri nets. They motivate their approach by arguing, as we do, that pure state-based or event-based formalisms lack expressiveness in important respects.

Huth *et al.* [HJS01] also propose a state/event framework, and define rich notions of abstraction and refinement. In addition, they provide ‘may’ and ‘must’ modalities for transitions, and show how to perform efficient three-valued verification on such structures. They do not, however, provide an automated CEGAR framework, and it is not clear whether they have implemented and tested their approach.

Giannakopoulou and Magee [GM03] define ‘fluent’ propositions within a labeled transition systems context to express action-based linear-time properties. A fluent proposition is a property that holds after it is initiated by an action and ceases to hold when terminated by another action. This work exploits partial-order reduction techniques and has been implemented in the LTSA tool.

In a comparatively early paper, De Nicola *et al.* [NFG93] propose a process algebraic framework with an action-based version of CTL as specification

formalism. Verification then proceeds by first translating the underlying labeled transition systems (LTSs) of processes into Kripke structures and the action-based CTL specifications into equivalent state-based CTL formulas. At that point, a model checker is used to establish or refute the property.

Dill [Dil88] defines ‘trace structures’ as algebraic objects to model both hardware circuits and their specifications. Trace structures can handle equally well states or events, although usually not both at the same time. Dill’s approach to verification is based on abstractions and compositional reasoning, albeit without an iterative counterexample-driven refinement loop.

In general, events (input signals) in circuits can be encoded via changes in state variables. Browne makes use of this idea in [Bro89], which features a CTL* specification formalism. Browne’s framework also features abstractions and compositional reasoning, in a manner similar to Dill’s.

Finally, Burch [Bur92] extends the idea of trace structures into a full-blown theory of ‘trace algebra’. The focus here however is the modeling of discrete and continuous time, and the relationship between these two paradigms. This work also exploits abstractions and compositionality, however once again without automated counterexample-guided refinements.

3 Labeled Kripke Structures

A labeled Kripke structure (LKS for short) is a 7-tuple $(S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ with S a finite set of *states*, $Init \subseteq S$ a set of initial states, P a finite set of *atomic state propositions*, $\mathcal{L} : S \rightarrow 2^P$ a *state-labeling function*, $T \subseteq S \times S$ a *transition relation*, Σ a finite set (*alphabet*) of *events* (or *actions*), and $\mathcal{E} : T \rightarrow (2^\Sigma \setminus \{\emptyset\})$ a *transition-labeling function*. We often write $s \xrightarrow{A} s'$ to mean that $(s, s') \in T$ and $A \subseteq \mathcal{E}(s, s')$.² In case A is a singleton set $\{a\}$ we write $s \xrightarrow{a} s'$ rather than $s \xrightarrow{\{a\}} s'$. Note that both states and transitions are ‘labeled’, the former with sets of atomic propositions, and the latter with non-empty sets of events. We further assume that our transition relation is *total* (every state has some successor), so that deadlock does not arise.

A *path* $\pi = \langle s_1, a_1, s_2, a_2, \dots \rangle$ of an LKS is an alternating infinite sequence of states and events subject to the following: for each $i \geq 1$, $s_i \in S$, $a_i \in \Sigma$, and $s_i \xrightarrow{a_i} s_{i+1}$.

The *language* of an LKS M , denoted $L(M)$, consists of the set of maximal paths of M whose first state lies in the set $Init$ of initial states of M .

3.1 Abstraction

Let $M = (S, Init, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and $A = (S_A, Init_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$ be two LKSs. We say that A is an *abstraction* of M , written $M \sqsubseteq A$, iff

² In keeping with standard mathematical practice, we write $\mathcal{E}(s, s')$ rather than the more cumbersome $\mathcal{E}((s, s'))$.

1. $P_A \subseteq P$,
2. $\Sigma_A = \Sigma$, and
3. For every path $\pi = \langle s_1, a_1, \dots \rangle \in L(M)$ there exists a path $\pi' = \langle s'_1, a'_1, \dots \rangle \in L(A)$ such that, for each $i \geq 1$, $a'_i = a_i$ and $\mathcal{L}_A(s'_i) = \mathcal{L}(s_i) \cap P_A$.

In other words, A is an abstraction of M if the ‘propositional’ language accepted by A contains the ‘propositional’ language of M , when restricted to the atomic propositions of A . This is similar to the well-known notion of ‘existential abstraction’ for Kripke structures in which certain variables are hidden [CGJ+00].

Two-way abstraction defines an equivalence relation $\sim$ on LKSs: $M \sim M'$ iff $M \sqsubseteq M'$ and $M' \sqsubseteq M$. We shall only be interested in LKSs up to $\sim$ -equivalence.

3.2 Parallel Composition

The notion of parallel composition we consider in this paper allows for communication through shared actions only; in particular, we forbid the sharing of variables. This restriction facilitates the use of compositional reasoning in verifying specifications.

Let $M_1 = (S_1, \text{Init}_1, P_1, \mathcal{L}_1, T_1, \Sigma_1, \mathcal{E}_1)$ and $M_2 = (S_2, \text{Init}_2, P_2, \mathcal{L}_2, T_2, \Sigma_2, \mathcal{E}_2)$ be two LKSs. M_1 and M_2 are said to be *compatible* if (i) they do not share variables: $S_1 \cap S_2 = P_1 \cap P_2 = \emptyset$, and (ii) their parallel composition (as defined below) yields a total transition relation (so that no deadlock can occur). The parallel composition of M_1 and M_2 (assumed to be compatible)³ is given by $M_1 \parallel M_2 = (S_1 \times S_2, \text{Init}_1 \times \text{Init}_2, P_1 \cup P_2, \mathcal{L}_1 \cup \mathcal{L}_2, T, \Sigma_1 \cup \Sigma_2, \mathcal{E})$, where $(\mathcal{L}_1 \cup \mathcal{L}_2)(s_1, s_2) = \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$, and T and $\mathcal{E}$ are such that $(s_1, s_2) \xrightarrow{A} (s'_1, s'_2)$ iff $A \neq \emptyset$ and one of the following holds:

1. $A \subseteq \Sigma_1 \setminus \Sigma_2$ and $s_1 \xrightarrow{A} s'_1$ and $s'_2 = s_2$,
2. $A \subseteq \Sigma_2 \setminus \Sigma_1$ and $s_2 \xrightarrow{A} s'_2$ and $s'_1 = s_1$, and
3. $A \subseteq \Sigma_1 \cap \Sigma_2$ and $s_1 \xrightarrow{A} s'_1$ and $s_2 \xrightarrow{A} s'_2$.

In other words, components must synchronize on shared actions and proceed independently on local actions. Moreover, local variables are preserved by the respective states of each component. This notion of parallel composition is derived from CSP; see also [ACFM85].

Let M_1 and M_2 be as above, and let $\pi = \langle (s_1^1, s_2^1), a_1, \dots \rangle$ be an alternating infinite sequence of states and events of $M_1 \parallel M_2$. The *projection* $\pi \upharpoonright M_i$ of π on M_i consists of the (possibly finite) subsequence of $\langle s_1^1, a_1, \dots \rangle$ obtained by simply removing all pairs $\langle a_j, s_{j+1}^i \rangle$ for which $a_j \notin \Sigma_i$. In other words, we keep from π only those states that belong to M_i , and excise any transition labeled with an event not in M_i ’s alphabet.

³ The assumption of deadlock-freedom greatly simplifies our exposition, and also enables us to use a wider class of abstractions. At the moment, the onus is on the user to ensure that all LKSs to be composed in parallel are compatible. In the future, we plan to incorporate an optional deadlock-freedom checker within MAGIC.

We now record the following theorem, which extends similar standard results for the process algebra CSP (for related proofs, we refer the reader to [Ros97]).

Theorem 1.

1. *Parallel composition is (well-defined and) associative and commutative up to $\sim$ -equivalence. Thus, in particular, no bracketing is required when combining more than two LKSs.*
2. *Let $M_1, \dots, M_n$ be compatible LKSs, and let $A_1, \dots, A_n$ be respective abstractions of the M_i : for each i , $M_i \sqsubseteq A_i$. Then $M_1 \parallel \dots \parallel M_n \sqsubseteq A_1 \parallel \dots \parallel A_n$. In other words, parallel composition preserves the abstraction relation.*
3. *Let $M_1, \dots, M_n$ be compatible LKSs with respective alphabets $\Sigma_1, \dots, \Sigma_n$, and let π be an infinite alternating sequence of states and events of the composition $M_1 \parallel \dots \parallel M_n$. Then $\pi \in L(M_1 \parallel \dots \parallel M_n)$ iff, for each i , there exists $\pi'_i \in L(M_i)$ such that $\pi \upharpoonright M_i$ is a prefix⁴ of π'_i . In other words, whether a path belongs to the language of a parallel composition of LKSs can be checked by projecting and examining the path on each individual component separately.*

Theorem 1 forms the basis of our compositional approach to verification: abstraction, counterexample validation, and refinement can all be done component-wise.

4 State/Event Linear Temporal Logic

We now present a logic enabling us to refer easily to both states and events when constructing specifications.

Given an LKS $M = (S, \text{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$, we consider linear temporal logic *state/event formulas* over the sets P and Σ (here p ranges over P and a ranges over Σ):

$$\phi ::= p \mid a \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{G}\phi \mid \mathbf{F}\phi \mid \phi \mathbf{U} \phi.$$

We write SE-LTL to denote the resulting logic, and in particular to distinguish it from (standard) LTL.

Let $\pi = \langle s_1, a_1, s_2, a_2, \dots \rangle$ be a path. π^i stands for the suffix of π starting in state s_i . We then inductively define path-satisfaction of SE-LTL formulas as follows:

1. $\pi \models p$ iff s_1 is the first state of π and $p \in \mathcal{L}(s_1)$,
2. $\pi \models a$ iff a is the first event of π ,
3. $\pi \models \neg\phi$ iff $\pi \not\models \phi$,
4. $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$,
5. $\pi \models \mathbf{X}\phi$ iff $\pi^2 \models \phi$,
6. $\pi \models \mathbf{G}\phi$ iff, for all $i \geq 1$, $\pi^i \models \phi$,

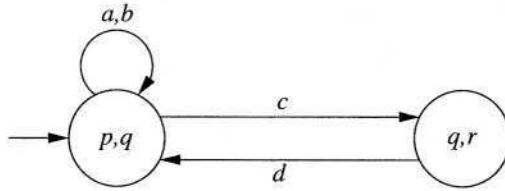
⁴ By convention, an infinite sequence is prefix of another one iff they are the same.

7. $\pi \models \mathbf{F}\phi$ iff, for some $i \geq 1$, $\pi^i \models \phi$, and
8. $\pi \models \phi_1 \mathbf{U} \phi_2$ iff there is some $i \geq 1$ such that $\pi^i \models \phi_2$ and, for all $1 \leq j \leq i-1$, $\pi^j \models \phi_1$.

We then let $M \models \phi$ iff, for every path $\pi \in L(M)$, $\pi \models \phi$.

We also use the derived **W** operator: $\phi_1 \mathbf{W} \phi_2$ iff $(\mathbf{G}\phi_1) \vee (\phi_1 \mathbf{U} \phi_2)$, as well as standard boolean connectives such as $\rightarrow$, etc.

As a simple example, consider the following LKS M . It has two states, the leftmost of which is the sole initial state. Its set of atomic state propositions is $\{p, q, r\}$; the first state is labeled with $\{p, q\}$ and the second with $\{q, r\}$. M 's transitions are similarly labeled with sets of events drawn from the alphabet $\{a, b, c, d\}$.



As the reader may easily verify, $M \models \mathbf{G}(c \rightarrow \mathbf{F}r)$ but $M \not\models \mathbf{G}(b \rightarrow \mathbf{F}r)$. Note also that $M \models \mathbf{G}(d \rightarrow \mathbf{F}r)$, but $M \not\models \mathbf{G}(d \rightarrow \mathbf{X}\mathbf{F}r)$.

4.1 Automata-Based Verification

We aim to reduce SE-LTL verification problems to standard automata-theoretic techniques for LTL. Note that a standard—but unsatisfactory—way of achieving this is to explicitly encode actions through changes in (additional) state variables, and then proceed with LTL verification. Unfortunately, this trick usually leads to a significant blow-up in the state space, and consequently yields much larger verification times. The approach we present here, on the other hand, does *not* alter the size of the LKS, and is therefore considerably more efficient.

We first recall some basic results about LTL, Kripke structures, and automata-based verification.

A Kripke structure is simply an LKS minus the alphabet and the transition-labeling function; as for LKSs, the transition relation of a Kripke structure is required to be total. An LTL formula is an SE-LTL formula which makes no use of events as atomic propositions.

For P a set of atomic propositions, let $\mathbf{B}_P \cong 2^{2^P}$ denote the set of boolean combinations of atomic propositions over P .

A Büchi automaton is a 6-tuple $B = (S_B, \text{Init}_B, P, \mathcal{L}_B, T_B, \text{Acc})$ with S_B a finite set of states, $\text{Init}_B \subseteq S_B$ a set of initial states, P a finite set of atomic state propositions, $\mathcal{L}_B : S_B \rightarrow \mathbf{B}_P$ a state-labeling function, $T \subseteq S_B \times S_B$ a transition relation, and $\text{Acc} \subseteq S_B$ a set of *accepting* states.

Note that the transition relation is not required to be total, and is moreover unlabeled. Note also that the states of a Büchi automaton are labeled with arbitrary boolean combinations of atomic propositions.

For π an infinite sequence of states of a Büchi automaton, let $\text{inf}(\pi) \subseteq S_B$ be the set of states which occur infinitely often in π . π is *accepted* by the Büchi automaton B if $\text{inf}(\pi) \cap \text{Acc} \neq \emptyset$. The set of all such accepted paths is written $L(B)$.

Let $M = (S, \text{Init}, P, \mathcal{L}, T)$ be a Kripke structure. The state-labeling function $\mathcal{L} : S \rightarrow 2^P$ indicates, for each state $s \in S$, exactly which atomic propositions hold at s ; such labeling is equivalent to asserting that the compound proposition $\bigwedge \mathcal{L}(s) \wedge \bigwedge \{\neg p \mid p \in P \setminus \mathcal{L}(s)\}$ holds at s . Let us denote this compound proposition by $\widetilde{\mathcal{L}(s)}$. Every Kripke structure can therefore be viewed as a Büchi automaton, where we consider every state to be accepting.

Let $B = (S_B, \text{Init}_B, P, \mathcal{L}_B, T_B, \text{Acc})$ be a Büchi automaton over the same set of atomic propositions as M . We can define the ‘standard’ product $M \times B = (S', \text{Init}', -, -, T', \text{Acc}')$ as a product of Büchi automata. More precisely,

1. $S' = \{(s, b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)} \text{ implies } \mathcal{L}_B(b)\}$,
2. $(s, b) \longrightarrow (s', b') \text{ iff } s \longrightarrow s' \text{ and } b \longrightarrow b'$,
3. $(s, b) \in \text{Init}' \text{ iff } s \in \text{Init} \text{ and } b \in \text{Init}', \text{ and}$
4. $(s, b) \in \text{Acc}' \text{ iff } b \in \text{Acc}$.

The non-symmetrical standard product $M \times B$ accepts exactly those paths of M which are ‘consistent’ with B . Its main technical use lies in the following result of Gerth *et al.* [GPVW95]:

Theorem 2. *Given a Kripke structure M and LTL formula ϕ , there is a Büchi automaton $B_{\neg\phi}$ such that*

$$M \models \phi \text{ iff } L(M \times B_{\neg\phi}) = \emptyset.$$

An efficient tool to convert LTL formulas into optimized Büchi automata with the above property is Somenzi and Bloem’s Wring [Wri,SB00].

We now turn to labeled Kripke structures. Let $M = (S, \text{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ be an LKS. Recall that SE-LTL formulas allow events in Σ to stand for atomic propositions. For $x \in \Sigma$, let us therefore write $\tilde{x}$ to denote the (formal) compound proposition $x \wedge \bigwedge \{\neg y \mid y \in \Sigma \setminus \{x\}\}$. We can also, given an SE-LTL formula ϕ over P and Σ , interpret ϕ as an LTL formula over $P \cup \Sigma$ (viewed as atomic *state* propositions); let us denote the latter formula by ϕ^b . ϕ^b is therefore syntactically identical to ϕ , but differs from ϕ in its semantic interpretation.

We now define the *state/event product* of a labeled Kripke structure with a Büchi automaton. Let M be as above, and let $B = (S_B, \text{Init}_B, P \cup \Sigma, \mathcal{L}_B, T_B, \text{Acc})$ be a Büchi automaton over the set of atomic state propositions $P \cup \Sigma$. The state/event product $M \otimes B = (S', \text{Init}', -, -, T', \text{Acc}')$ is a Büchi automaton that satisfies

1. $S' = \{(s, b) \in S \times S_B \mid \widetilde{\mathcal{L}(s)} \text{ implies } \exists \Sigma. \mathcal{L}_B(b)\}^5$,

⁵ The term $\exists \Sigma. \mathcal{L}_B(b)$ denotes the formula $\mathcal{L}_B(b)$ in which all atomic Σ -propositions have been existentially quantified out; in practice, however, the output of Wring is presented in a format which renders this operation trivial (and computationally inexpensive).

2. $(s, b) \longrightarrow (s', b')$ iff there exists $x \in \Sigma$ such that $s \xrightarrow{x} s'$ and $b \longrightarrow b'$ and $(\mathcal{L}(s) \wedge \tilde{x})$ implies $\mathcal{L}_B(b)$,
3. $(s, b) \in \text{Init}'$ iff $s \in \text{Init}$ and $b \in \text{Init}_B$, and
4. $(s, b) \in \text{Acc}'$ iff $b \in \text{Acc}$.

Finally, we have:

Theorem 3. *For any LKS M and SE-LTL formula ϕ ,*

$$M \models \phi \text{ iff } L(M \otimes B_{\neg\phi^b}) = \emptyset.$$

Note that the state/event product does *not* require an enlargement of the LKS M (although we consider below just such an enlargement in the course of the proof of the theorem).

Proof. Observe that a state of M can have several differently-labeled transitions emanating from it. However, by duplicating states (and transitions) as necessary, we can transform M into a $\sim$ -equivalent LKS M' having the following property: for every state s of M' , the transitions emanating from s are all labeled with the same (single) event. As a result, the validity of an SE-LTL atomic event proposition a in a given state of M' does not depend on the particular path to be taken from that state, and can therefore be recorded as a propositional state variable of the state itself. Formally, this gives rise to a Kripke structure M'' over atomic state propositions $P \cup \Sigma$.

We now claim that

$$L(M \otimes B_{\neg\phi^b}) = \emptyset \text{ iff } L(M'' \times B_{\neg\phi^b}) = \emptyset. \quad (1)$$

To see this, notice first that there is a bijection between $L(M)$ and $L(M'')$ (which we denote $\pi \mapsto \pi''$). Next, observe that any path in $L(M \otimes B_{\neg\phi^b})$ can be decomposed as a pair (π, β) , where $\pi \in L(M)$ and $\beta \in L(B_{\neg\phi^b})$; likewise, any path in $L(M'' \times B_{\neg\phi^b})$ can be decomposed as a pair (π'', β) , where $\pi'' \in L(M'')$ and $\beta \in L(B_{\neg\phi^b})$. A straightforward inspection of the relevant definitions then reveals that $(\pi, \beta) \in L(M \otimes B_{\neg\phi^b})$ iff $(\pi'', \beta) \in L(M'' \times B_{\neg\phi^b})$, which establishes our claim.

Finally, we clearly have $M \models \phi$ iff $M' \models \phi$ iff $M'' \models \phi^b$. Combining this with Theorem 2 and Equation 1 above, we get $M \models \phi$ iff $L(M \otimes B_{\neg\phi^b}) = \emptyset$, as required. $\square$

The significance of Theorem 3 is that it enables us to make use of the highly optimized algorithms and tools available for verifying LTL formulas on Kripke structures to verify *SE-LTL* specifications on *labeled* Kripke structures, at no additional cost.

5 A Surge Protector

We describe a safety-critical current surge protector in order to illustrate the advantages of state/event-based implementations and specifications over both the pure state-based and the pure event-based approaches.

The surge protector is meant at all times to disallow changes in current beyond a varying threshold. The labeled Kripke structure in Figure 1 captures the main functional aspects of such a protector in which the possible values of the current and threshold are 0, 1, and 2. The threshold value is stored in the variable m , and changes in threshold and current are respectively communicated via the events $m0$, $m1$, $m2$, and $c0$, $c1$, $c2$.⁶ Note, for instance, that when $m = 1$ the protector accepts changes in current to values 0 and 1, but not 2 (in practice, an attempt to hike the current up to 2 should trigger, say, a fuse and a jump to an emergency state, behaviors which are here abstracted away).

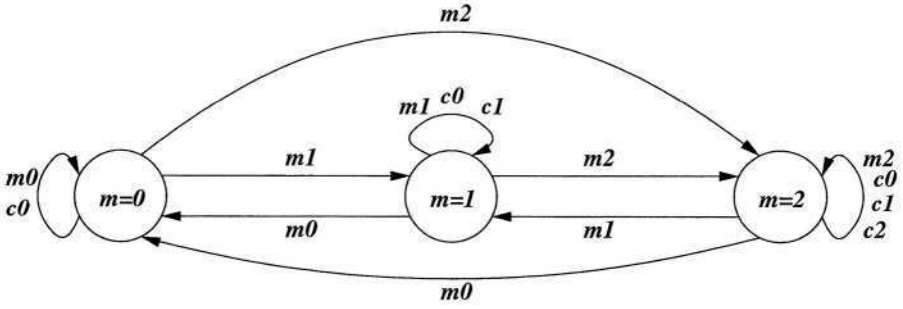


Fig. 1. The LKS of a surge protector

The required specification is neatly captured as the following SE-LTL formula:

$$\phi_{se} = \mathbf{G}((c2 \rightarrow m = 2) \wedge (c1 \rightarrow (m = 1 \vee m = 2))).$$

By way of comparison, Figure 2 represents the (event-free) Kripke structure that captures the same behavior as the LKS of Figure 1. In this pure state-based formalism, nine states are required to capture all the reachable combinations of threshold ($m = i$) and last current changes ($c = j$) values.

The data (9 states and 39 transitions) compares unfavorably with that of the LKS in Figure 1 (3 states and 9 transitions). Moreover, as the allowable current ranges increase, the number of states of the LKS will grow linearly, as opposed to quadratically for the Kripke structure. The number of transitions of both will grow quadratically, but with a roughly four-fold larger factor for the Kripke structure. These observations highlight the advantages of a state/event approach, which of course will be more or less pronounced depending on the type of system under consideration.

⁶ The reader may object that we have only allowed for *boolean* variables in our definition of labeled Kripke structures; it is however trivial to implement more complex types, such as bounded integers, as boolean encodings, and we have therefore elided such details here.

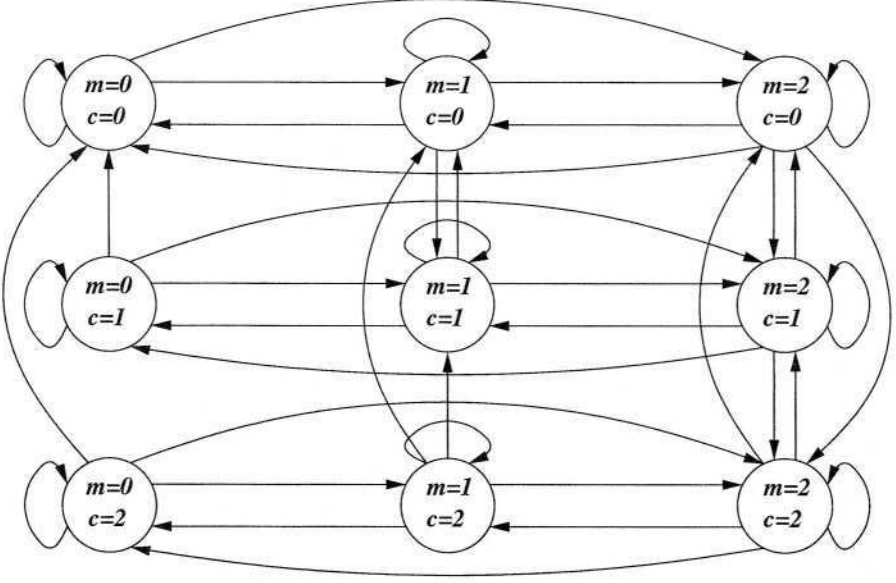


Fig. 2. The Kripke structure of a surge protector

Another advantage of the state/event approach is witnessed when one tries to write down specifications. In this instance, the specification we require is

$$\phi_s = \mathbf{G}(((c = 0 \vee c = 2) \wedge \mathbf{X}(c = 1)) \rightarrow (m = 1 \vee m = 2)) \wedge \\ \mathbf{G}(((c = 0 \vee c = 1) \wedge \mathbf{X}(c = 2)) \rightarrow m = 2),$$

which is arguably significantly more complex than ϕ_{se} .

The pure event-based specification ϕ_e capturing the same requirement is also clearly more complex than ϕ_{se} :

$$\phi_e = \mathbf{G}(m0 \rightarrow ((\neg c1) \mathbf{W} (m1 \vee m2))) \wedge \\ \mathbf{G}(m0 \rightarrow ((\neg c2) \mathbf{W} m2)) \wedge \\ \mathbf{G}(m1 \rightarrow ((\neg c2) \mathbf{W} m2)).$$

The greater simplicity of the implementation and specification associated with the state/event formalism is not purely a matter of aesthetics, or even a safeguard against subtle mistakes; experiments also suggest that the state/event formulation yields significant gains in both time and memory during verification. We implemented three parameterized instances of the surge protector as simple C programs, in one case allowing message passing (representing the LKS), and in the other relying solely on local variables (representing the Kripke structure). We also wrote corresponding specifications respectively as SE-LTL and LTL formulas (as above) and converted these into Büchi automata using the tool Wring [Wri]. Figure 3 records the number of Büchi states and transitions associated with

the specification, as well as the time taken by MAGIC to construct the Büchi automaton and confirm that the corresponding implementation indeed meets the specification.

Range	Pure State				Pure Event				State/Event			
	St	Tr	B-T	T-T	St	Tr	B-T	T-T	St	Tr	B-T	T-T
2	4	5	253	383	6	10	245	320	3	4	184	252
3	8	12	270	545	12	23	560	674	4	6	298	407
4	14	23	492	1141	20	41	1597	1770	5	8	243	391
5	22	38	1056	2326	30	64	3795	4104	6	10	306	497
6	32	57	2428	4818	42	92	12077	12660	7	12	614	962
7	44	80	6249	10358	56	125	54208	55064	8	14	930	1321
8	58	107	17503	24603	72	163	372784	374166	9	16	2622	3133
9	74	138	55950	67553	*	*	*	*	10	18	8750	9488
10	92	173	195718	213969	*	*	*	*	11	20	33556	34503
11	*	*	*	*	*	*	*	*	12	22	135252	136500
12	*	*	*	*	*	*	*	*	13	24	534914	536451
13	*	*	*	*	*	*	*	*	*	*	*	*

Fig. 3. Comparison of pure state-based, pure event-based and state/event-based formalisms. Values of c and m range between 0 and **Range**. **St** and **Tr** respectively denote the number of states and transitions of the Büchi automaton corresponding to the specification. **B-T** is the Büchi construction time and **T-T** is the total verification time. All times are reported in milliseconds. A * indicates that the Büchi automaton construction did not terminate in 10 minutes.

A careful inspection of the table in Figure 3 reveals several consistent trends. First, the number of Büchi states increases quadratically with the value of *Range* for both the pure state-based and pure event-based formalisms. In contrast, the increase is only linear when both states and events are used. We notice a similar pattern among the number of transitions in the Büchi automata. The rapid increase in the sizes of Büchi automata will naturally contribute to increased model checking time. However, we notice that the major portion of the total verification time is required to construct the Büchi automaton. While this time increases rapidly in all three formalisms, the growth is observed to be most benign for the state/event scenario. The net result is clearly evident from Figure 3. Using both states and events allows us to push the limits of c and m beyond what is possible by using either states or events alone.

6 Compositional Counterexample-Guided Verification

We now discuss how our framework enables us to verify SE-LTL specifications on parallel compositions of labeled Kripke structures incrementally and compositionally.

When trying to determine whether an SE-LTL specification holds on a given LKS, the following result is the key ingredient needed to exploit abstractions in the verification process:

Theorem 4. *Let M and A be LKSs with $M \sqsubseteq A$. Then for any SE-LTL formula ϕ over M which mentions only propositions (and events) of A ,*

$$\text{if } A \models \phi \text{ then } M \models \phi.$$

Proof. This follows easily from the fact that every path of M is matched by a corresponding property-preserving path of A . $\square$

Let us now assume that we are given a collection $M_1, \dots, M_n$ of LKSs, as well as an SE-LTL specification ϕ , with the task of determining whether $M_1 \parallel \dots \parallel M_n \models \phi$. We first create initial abstractions $A_1 \supseteq M_1, \dots, A_n \supseteq M_n$, in a manner to be discussed shortly. We then check whether $A_1 \parallel \dots \parallel A_n \models \phi$. In the affirmative, we conclude (by Theorems 1 and 4) that $M_1 \parallel \dots \parallel M_n \models \phi$ as well. In the negative, we are provided with a counterexample $\pi_A \in L(A_1 \parallel \dots \parallel A_n)$ such that $\pi_A \not\models \phi$. We must then determine whether this counterexample is real or spurious, i.e., whether it corresponds to a counterexample $\pi \in L(M_1 \parallel \dots \parallel M_n)$.

This validation check can be performed compositionally, as follows. According to Theorem 1, the counterexample is real iff for each i , the projection $\pi_A \upharpoonright A_i$ corresponds to (the prefix of) a valid behavior of M_i . To this end, we ‘simulate’ $\pi_A \upharpoonright A_i$ on M_i . If M_i accepts the path, we go on to the next component. Otherwise, we *refine* our abstraction A_i , yielding a new abstraction A'_i with $M_i \sqsubseteq A'_i \sqsubseteq A_i$ and such that A'_i also rejects the projection $\pi_A \upharpoonright A'_i$ of the spurious counterexample π_A .

This process is iterated until either the specification is proved, or a real counterexample is found. Termination follows from the fact that the LKSs involved are all finite, and therefore admit only finitely many distinct abstractions.⁷

The advantage of this approach is that all the abstractions that we consider in this paper are *existential abstraction quotients* of LKSs. In other words, abstractions are obtained by lumping together states of the original LKSs, and have therefore smaller state spaces. Formally, given an LKS $M = (S, \text{Init}, P, \mathcal{L}, T, \Sigma, \mathcal{E})$ and a partition $\approx$ of the state space S , an *existential abstraction quotient* of M is any LKS $A = (S_A, \text{Init}_A, P_A, \mathcal{L}_A, T_A, \Sigma_A, \mathcal{E}_A)$ such that

1. $S_A = S/\approx$,
2. $\text{Init}_A = \{[s] \in S/\approx \mid \exists s' \in [s]. s' \in \text{Init}\}$,
3. $P_A \subseteq P$, and for all $s, s' \in S$, if $s \approx s'$ then $\mathcal{L}(s) \cap P_A = \mathcal{L}(s') \cap P_A$,
4. for all $s \in S$, $\mathcal{L}_A([s]) = \mathcal{L}(s) \cap P_A$,

⁷ When the LKSs $M_1, \dots, M_n$ are generated from C programs via predicate abstraction, as is the case for MAGIC, termination will depend on whether MAGIC is eventually able to generate sufficiently strong predicates. Although this in general cannot be guaranteed, as a result of the undecidability of the halting problem, in practice it has not been observed to cause any problems.

5. $\Sigma_A = \Sigma$, and
6. for all $s, s' \in S$ and $a \in \Sigma$, $[s_1] \xrightarrow{a} [s_2]$ iff there exists $s'_1 \in [s_1]$, $s'_2 \in [s_2]$ such that $s'_1 \xrightarrow{a} s'_2$.

We now have:

Theorem 5. *For M an LKS and $\approx$ a partition of S , any existential abstraction quotient A of M as defined above is a genuine abstraction of M in the sense of Section 3.1: $M \sqsubseteq A$.*

Note that an abstraction of M is entirely determined by the partition $\approx$ and the set P_A of atomic state propositions. In our case, given an SE-LTL formula ϕ , we shall fix P_A to be $\text{stateprop}(\phi)$, the set of all atomic state propositions appearing in ϕ . Abstractions of M can therefore be identified with partitions $\approx$ of S that meet condition 3 above; we denote the corresponding abstraction by $M/\approx$.

Theorem 6. *Let M be an LKS and let $M/\approx$ be an abstraction of M . For any refinement $\approx'$ of the partition $\approx$, $M/\approx'$ is an abstraction of M that is also a refinement of $M/\approx$: $M \sqsubseteq M/\approx' \sqsubseteq M/\approx$.*

We leave the proofs of Theorems 5 and 6 to the reader.

To define the initial abstraction $M/\approx_1$, we let $s \approx_1 s'$ iff $\mathcal{L}(s) \cap \text{stateprop}(\phi) = \mathcal{L}(s') \cap \text{stateprop}(\phi)$ and $\text{enabled}(s) = \text{enabled}(s')$, where $\text{enabled}(s)$ denotes the set of actions that appear in transitions originating from s , etc.

We must refine our abstraction whenever we encounter a spurious counterexample $\pi_{A_k} \in L(M/\approx_k) \setminus L(M)$. We achieve this, in fully automated fashion, by constructing a refinement $\approx_{k+1}$ of the partition $\approx_k$ which splits abstract states along the path π_{A_k} . The approach we take is very similar to that presented in [COYC03]; unfortunately, the details involved are too lengthy to reproduce here, and we refer the reader to that paper for a thorough account of the technique.

As discussed above, MAGIC iterates this abstraction-validation-refinement procedure component-wise until the property ϕ of interest is either established ($M_1 \parallel \dots \parallel M_n \models \phi$) or refuted ($M_1 \parallel \dots \parallel M_n \not\models \phi$).

7 Experimental Results

We experimented with two broad sets of benchmarks. All our experiments were performed on an AMD Athlon XP 1600+ machine with 900 MB RAM running RedHat Linux 7.1. The first set of our examples were based on OpenSSL-0.9.6c, an open-source implementation of the SSL protocol. This is a popular protocol used for secure exchange of sensitive information over untrusted networks. SSL involves an initial handshake between a client and a server that attempt to establish a secure channel between themselves. The target of our verification process was the implementation of this handshake, comprising of about 350 lines of ANSI C code each for the server and the client.

From the official SSL specification [SSL] we derived a set of nine properties that every correct SSL implementation should satisfy. The first five properties are

Name	St(B)	Tr(B)	St(Mdl)	T(BA)	T(Mdl)	T(Ver)	T(Total)	Mem
svr-1-ss	4	5	5951	213	32195	1654	34090	-
svr-1-se	3	4	4269	209	18116	1349	19674	-
svr-2-ss	11	23	4941	292	31331	2479	34102	-
svr-2-se	3	4	4269	196	17897	1317	19410	-
svr-3-ss	37	149	5065	1147	26958	4031	32137	-
svr-3-se	3	4	4269	462	17950	1908	20319	-
svr-4-ss	16	41	5446	806	29809	7382	39341	28.6
svr-4-se	7	14	4333	415	21453	3513	25906	24.1
svr-5-ss	25	47	7951	690	48810	6842	56888	39.3
svr-5-se	20	45	4331	497	18808	2925	22765	24.2
clnt-1-ss	16	41	4867	793	24488	1235	26953	25.8
clnt-1-se	7	14	3693	376	17250	583	18683	22.1
clnt-2-ss	25	47	7574	699	43592	1649	46444	38.1
clnt-2-se	18	40	3691	407	15304	1087	17269	21.2
ssl-1-ss	25	47	24799528	874	65585	*	*	850.5
ssl-1-se	20	45	13558984	655	33091	2172139	2206983	162.4
ssl-2-ss	25	47	32597042	836	66029	*	*	346.6
ssl-2-se	18	40	15911791	713	34641	4148550	4185068	320.7
UCOS-BUG	8	14	873	205	3409	261	3880	-
UCOS-1	8	14	873	194	3365	2797	6357	-
UCOS-2	5	8	873	123	3372	2630	6127	-

Fig. 4. Experimental results with OpenSSL and Micro-C OS. **St(B)** and **Tr(B)** = respectively the number of states and transitions in the Büchi automaton; **St(Mdl)** = number of states in the model; **T(Mdl)** = model construction time; **T(BA)** = Büchi construction time; **T(Ver)** = model checking time; **T(Total)** = total verification time. All reported times are in milliseconds. **Mem** is the total memory requirement in MB. A * indicates that the model checking did not terminate within 2 hours and was aborted. In such cases, other measurements were made at the point of forced termination. A - indicates that the corresponding measurement was not taken.

relevant only to the server, the next two apply only to the client, and the last two properties refer to both a server and a client executing concurrently. For instance, the first property states that whenever the server asks the client to terminate the handshake, it eventually either gets a correct response from the client or exits with an error code. The second property expresses the fact that whenever the server receives a handshake request from a client, it eventually acknowledges the request or returns with an error code. The third property states that a server never exchanges encryption keys with a client once the cipher scheme has been changed.

Each of these properties were then expressed in SE-LTL, once using only states and again using both states and events. Table 4 summarizes the results of our experiments with these benchmarks. The SSL benchmarks have names of the form x-y-z where x denotes the type of the property and can be either *svr*,

$cInt$ or ssl , depending on whether the property refers respectively to only the server, only the client, or both server and client. y denotes the property number while z denotes the specification style and can be either ss (only states) or se (both states and events). We note that in each case the numbers for state/event properties are considerably better than those for the corresponding pure-state properties.

The second set of our benchmarks were obtained from the source code of version 2.00 of Micro-C OS. This is a popular, lightweight, real-time, multi-tasking operating system written in about 3000 lines of ANSI C. The OS uses a lock to ensure mutual exclusion for critical section code. Using SE-LTL we expressed two properties of the OS: (i) the lock is acquired and released alternately starting with an acquire and (ii) every time the lock is acquired it is eventually released. These properties were expressed using only events.

We found a **bug** in the OS that causes it to violate the first property. We informed the developers of the OS about this bug and were told that it has been detected and fixed. The developers also kindly supplied us with the latest source code for the OS, and we are currently attempting to find errors in it. The second property was found to be valid. In Figure 4 these experiments are named UCOS-BUG and UCOS-2 respectively. Next we fixed the bug and verified that the first property holds for the corrected OS. This experiment is called UCOS-1 in Figure 4.

8 Conclusion and Future Work

In this paper, we have presented an expressive framework for modeling and verifying linear-time temporal specifications on concurrent software systems. Our approach involves both states and events, and is predicated on a compositional counterexample-guided abstraction refinement scheme. We have also shown how standard automata-theoretic techniques for verifying linear temporal logic formulas can be ported to our framework at no extra cost, and have implemented these on our C model checker MAGIC. We have also carried out a number of experiments on the source code for OpenSSL-0.9.6c and Micro-C OS version 2, discovering a bug in the latter. These experiments have led us to conclude that not only does a state/event formalism facilitate the formulation of appropriate specifications (as compared to standard pure state-based or event-based frameworks), but also yield significant improvements in both verification time and memory usage.

There remain many avenues for further research. One is to consider alternative specification formalisms, such branching-time temporal logics. In our current framework, it may be possible to further optimize the automata-theoretic part of the verification, by directly transforming SE-LTL formulas into *labeled* Büchi automata. Doing so should yield more compact automata-based representations of specifications, resulting in a smaller overall state space. Another direction is to investigate other, more aggressive (and perhaps specification-dependent), notions

of abstraction. We are also currently working on a compositional CEGAR-based algorithm to check deadlock-freedom.

MAGIC is at present an explicit model checking tool—it could be worthwhile to incorporate symbolic and partial order techniques to improve its efficiency further. Another interesting area of research is to develop mechanisms to handle shared variables. Other modifications under consideration include the modeling of fairness constraints. Lastly, we are currently attempting to model and verify the source code of a controller for a large industrial metal-casting plant.

References

- [ACFM85] T. S. Anantharaman, E. M. Clarke, M. J. Foster, and B. Mishra. Compiling path expressions into VLSI circuits. In *Proceedings of POPL*, pages 191–204, 1985.
- [BLA] BLAST website, <http://www-cad.eecs.berkeley.edu/~rupak/blast>.
- [BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proceedings of CAV*, volume 1427, pages 319–331. Springer LNCS, 1998.
- [BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 203–213, 2001.
- [BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN*, volume 2057, pages 103–122. Springer LNCS, 2001.
- [Bro89] M. C. Browne. *Automatic verification of finite state machines using temporal logic*. PhD thesis, Carnegie Mellon University, 1989. Technical report no. CMU-CS-89-117.
- [BS01] J. Bradfield and C. Stirling. *Modal Logics and Mu-Calculi : An Introduction*, pages 293–330. Handbook of Process Algebra. Elsevier, 2001.
- [Bur92] J. Burch. *Trace algebra for automatic verification of real-time concurrent systems*. PhD thesis, Carnegie Mellon University, 1992. Technical report no. CMU-CS-92-179.
- [CCG⁺03] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proceedings of ICSE 2003*, pages 385–395, 2003.
- [CCK⁺02] P. Chauhan, E. M. Clarke, J. H. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Proceedings of FMCAD*, pages 33–51, 2002.
- [CDH⁺00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of ICSE*, pages 439–448. IEEE Computer Society, 2000.
- [CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

- [CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
- [CGJ⁺00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
- [CGKS02] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Shrichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *Proceedings of CAV*, pages 265–279, 2002.
- [CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, December 1999.
- [CGP03] J. M. Cobleigh, D. Giannakopoulou, and C. S. Păsăreanu. Learning assumptions for compositional verification. In *Proceedings of TACAS*, volume 2619, pages 331–346. Springer LNCS, 2003.
- [COYC03] S. Chaki, J. Ouaknine, K. Yorav, and E. M. Clarke. Automated compositional abstraction refinement for concurrent C programs: A two-level approach. In *Proceedings of SoftMC 03*. ENTCS 89(3), 2003.
- [Dil88] D. L. Dill. *Trace theory for automatic hierarchical verification of speed-independent circuits*. PhD thesis, Carnegie Mellon University, 1988. Technical report no. CMU-CS-88-119.
- [GL94] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Trans, on Programming Languages and Systems*, 16(3):843–871, 1994.
- [GM03] D. Giannakopoulou and J. Magee. Fluent model checking for event-based systems. In *Proceedings of FSE*. ACM Press, 2003.
- [GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
- [HJMQ03] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *Proceedings of CAV*, volume 2725. Springer LNCS, 2003.
- [HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of POPL*, pages 58–70, 2002.
- [HJS01] M. Huth, R. Jagadeesan, and D. Schmidt. Modal transition systems: A foundation for three-valued program analysis. In *Proceedings of ESOP 01*. LNCS 2028, 2001.
- [Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [HQR00] T. A. Henzinger, S. Qadeer, and S. K. Rajamani. Decomposing refinement proofs using assume-guarantee reasoning. In *Proceedings of ICCAD*, pages 245–252. IEEE Computer Society Press, 2000.
- [Koz83] D. Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.
- [Kur94] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
- [KV98] E. Kindler and T. Vesper. ESTL: A temporal logic for events and states. In *Proceedings of ATPN 98*, pages 365–383. LNCS 1420, 1998.
- [LBBO01] Y. Lakhnech, S. Berisalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Proceedings of TACAS*, volume 2031, pages 98–112. Springer LNCS, 2001.

- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of POPL*, 1985.
- [MAG] MAGIC website, <http://www.cs.cmu.edu/~chaki/magic>.
- [McM97] K. L. McMillan. A compositional rule for hardware design refinement. In *Proceedings of CAV*, volume 1254, pages 24–35. Springer LNCS, 1997.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, London, 1989.
- [NCOD97] G. Naumovich, L. A. Clarke, L. J. Osterweil, and M. B. Dwyer. Verification of concurrent software with FLAVERS. In *Proceedings of ICSE*, pages 594–595. ACM Press, 1997.
- [NFG93] R. De Nicola, A. Fantechi, S. Gnesi, and G. Ristori. An action-based framework for verifying logical and behavioural properties of concurrent systems. *Computer Networks and ISDN Systems*, 25(7):761–778, 1993.
- [NV95] R. De Nicola and F. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM (JACM)*, 42(2):458–487, 1995.
- [PDV01] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counterexamples when model checking abstracted Java programs. In *Proceedings of TACAS*, volume 2031, pages 284–298. Springer LNCS, 2001.
- [Pnu86] A. Pnueli. Application of temporal logic to the specification and verification of reactive systems: A survey of current trends. In J.W. de Bakker, W. P. de Roever, and G. Rozenburg, editors, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer, 1986.
- [QS81] J.P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *proceedings of Fifth Intern. Symposium on Programming*, pages 337–350, 1981.
- [Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, London, 1997.
- [SB00] F. Somenzi and R. Bloem. Efficient Büchi automata from LTL formulae. In *Computer-Aided Verification*, pages 248–263, 2000.
- [SLA] SLAM website, <http://research.microsoft.com/slam>.
- [SSL] OpenSSL. <http://wp.netscape.com/eng/ssl3/ssl-toc.html>.
- [Sto02] S. D. Stoller. Model-checking multi-threaded distributed Java programs. *International Journal on Software Tools for Technology Transfer*, 4(1):71–91, 2002.
- [Wri] Wring website, <http://vlsi.Colorado.edu/~rbloem/wring.html>.

Formalising Behaviour Trees with CSP

Kirsten Winter

School of Information Technology and Electrical Engineering
University of Queensland 4072, Australia
phone: +61 7 3365 1625 fax: +61 7 3365 4999
kirsten@svrc.uq.edu.au

Abstract. Behaviour Trees is a novel approach for requirements engineering. It advocates a graphical tree notation that is easy to use and to understand. Individual requirements are modelled as single trees which later on are integrated into a model of the system as a whole. We develop a formal semantics for a subset of Behaviour Trees using CSP. This work, on one hand, provides tool support for Behaviour Trees. On the other hand, it builds a front-end to a subset of the CSP notation and gives CSP users a new modelling strategy which is well suited to the challenges of requirements engineering.

Keywords: Requirements engineering, model checking, Behaviour Trees, CSP.

1 Introduction

Modelling system requirements in a complete and traceable manner is an essential step in system design. Usually, this step has to bridge the gap between a natural language description and a formal or informal notation. To ease the task, the notation should support the most direct translation from the given description. It should be easily understood by customers who are not familiar with mathematical notations. Ideally, it would also provide a means to trace back the ingredients in the resulting model to parts of the given text. Analysing the requirements model is a crucial step toward early error detection. Gaps and inconsistencies in the requirements discovered in the early phase of modelling can still be rectified easily. For larger systems, this analysis should be supported by tools.

Tool support suggests the use of a formal modelling notation. However, formal notations are usually not very close to informally given requirements and for customers are often hard to read and to understand. Addressing this twofold need, we suggest the integration of a graphical notation that supports requirements engineering, with a formal notation that provides a formal semantics and tool support for the analysis. We are aiming at integrating *Behaviour Trees* and *CSP*.

The Behaviour Tree Notation [Dro03] is a graphical notation that allows the user to first model individual requirements that are subsequently integrated into

a system design model. This integration is based on the tree structure: individual requirements are modelled by simple tree structures and are integrated by grafting one tree, A, onto a node, B, of another tree when the root node of A matches the node B. This tree model takes all components of the system into perspective within the same view, thus reflecting the natural language description. A view on the behaviour of a single component can later be factored out from the integrated behaviour tree, as well as the structural view of the system's architecture. Moreover, the notation supports the bookkeeping of modelling information. So far, there is no formal semantics defined for this notation.

Communicating Sequential Processes (CSP) [Hoa85,Ros98] is a process algebra for elegantly specifying the behaviour of interacting components. It is well suited to reflect the semantics of Behaviour Trees because the language provides all needed constructs for modelling the variants of control flow used in Behaviour Tree models. The model checker FDR (Failure Divergence Refinement) [For96] provides an analysis tool for CSP and hence can be used for analysing Behaviour Tree models if we provide a translation from the latter into CSP. It allows the user to check a model for deadlock and livelock and for the refinement relation between two models. These checks can be exploited to check a requirements model for inconsistencies and incompleteness.

Interacting CSP processes, on the other hand, synchronised via the CSP channel mechanism can be challenging to read if the user is faced with a large number of components that interact a lot. This dictates another motivation for the integration: Behaviour Trees make a nice graphical front-end for representing the interaction of CSP processes. Moreover, Behaviour Trees provide a systematic and constructive way of capturing functional requirements in a system design model. A similar stepwise approach could not easily be followed when using the CSP notation to model functional requirements. Points of integration for two individual requirements would be difficult to determine in a CSP setting.

Similar work has been undertaken by others (see e.g., [NB02,BD00]) by integrating parts of UML and CSP. Although the integration step is different, the motivation is quite similar: CSP serves as formal semantics to a non-formal graphical notation, and the graphical notation provides a user-friendly front-end for CSP. In extension to that, our approach adds a new modelling dimension to the process algebra.

The paper is organised as follows: Section 2 introduces the notation of Behaviour Trees. Section 3 briefly overviews the CSP notation and describes the integration of Behaviour Trees and CSP. The integration is illustrated by means of an example in Section 4. In Section 5, we summarise the results of our analysis using FDR. Section 6 summarises this work and gives an outlook to future work.

2 Behaviour Trees

A great challenge of requirements engineering is how to get from a set of functional requirements to a system design that meets these requirements. The task is even harder if the requirements show defects and will subsequently change. Be-

haviour Trees [Dro03] is a new notation that targets this challenge by promoting a constructive and systematic way for going from a set of functional requirements to a design that satisfies those requirements. Behaviour is expressed in terms of components realising *states*, undergoing *events* and satisfying *constraints* which determine *control flow* and *data flow*. Moreover components may have *threads* of concurrent behaviour. These constituents are the set of key elements of the Behaviour Tree Notation as shown in Figure 1.

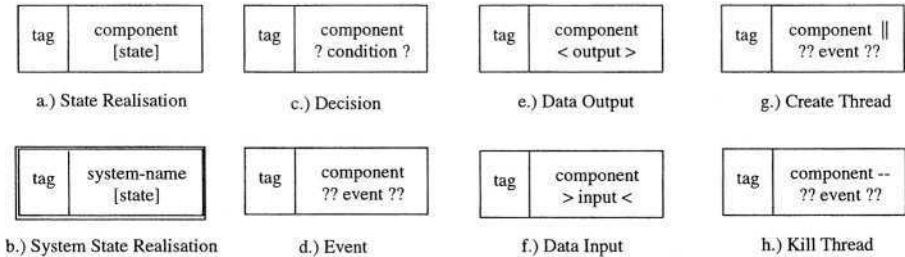


Fig. 1. Key Elements of the Behaviour Tree Notation

A box refers to a component and either its state (Fig.1.a and b), its condition on the control-flow (Fig.1.c), an event occurrence (Fig.1.d and g) , or input-/output-flow (Fig.1.e and f). Using a special construct (Fig.1.h) we can also model the termination of a thread. Behaviour of the system component is distinguished through a double-framed box. The boxes are the nodes of the tree. They also carry a *tag* which is a pointer to the part of the requirements that is modelled by the (sub-)tree (usually a sentence). Additionally, tags can have a ‘+’ indicating that this box models an assumption that was implicit in the requirements text or a ‘-’ for indicating that this information is actually missing in the informal requirements. This notational convention maximises traceability from the model back to the original text. A ‘++’ in the tag-frame is used for changed requirements; this helps developers managing the bookkeeping for the evolution of the system.

In contrast to other notations such as sequence diagrams [HD99], activity charts [BRJ99], and Statecharts [Har87], one Behaviour Tree can capture the behaviour of a number of components. A tree comprises boxes referring to multiple components and modelling the causal dependencies of their control flow. This allows a direct mapping from a natural language description into a tree structure. The Behaviour Tree model of the requirements is built sentence by sentence. For instance, the description *when the door is open the light should go on* is translated into the tree shown in Figure 2. The arrows that link the boxes in a tree-like manner denote the control flow and the causal dependencies between the components. We distinguish the following different forms (as depicted in Figure 3, note that the number of branches is not restricted to two):

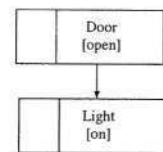


Fig. 2. Example tree

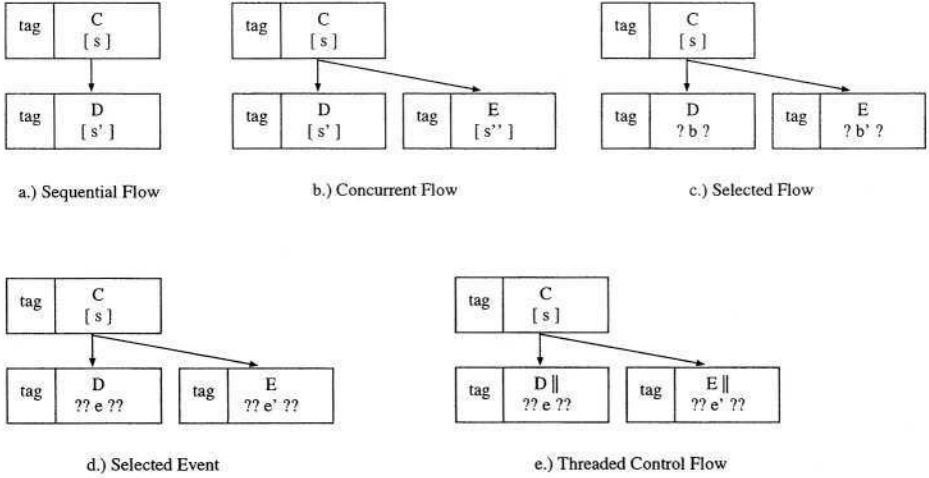


Fig. 3. Syntax of the Control Flow

- a) Sequential Flow: Component C realises state s and sequentially passes control to component D which then realises state s' .
- b) Concurrent Flow: Component C realises state s and concurrently passes control to components D and E.
In some cases (e.g., see Figure 6), the control flow of the tree proceeds only after one of the boxes (e.g., only the box $D[s']$ has an outgoing edge). It means the two components realise their states s' and s'' concurrently and after that the system continues in a sequential manner.
- c) Selected Flow: On receiving control from component C, component D passes control to its successor if the boolean condition b is true; E passes on control if b' is true. (The notation does not enforce the conditions to exclude each other or the cases to be complete. It is part of the later analysis of the model to ensure these criteria.)
- d) Selected Event: On receiving control from component C, component D passes control to its successor if event e occurs, if event e' occurs component E passes on control. If both events occur simultaneously the flow of control will be chosen non-deterministically.
- e) Threaded Control Flow: On receiving control from component C, both events e and e' trigger independent threads; one event occurring before the other does not extinguish the possibility of the other event occurring and starting the other thread. This notation is introduced in order to distinguish concurrent flow guided by events from the flow of selected events (as in Figure 3(d)). A thread can also be killed by another thread (the notation is shown in Figure 1(h)).

Modelling the given requirements as Behaviour Trees happens in a stepwise manner: each sentence (or set of sentences which address the same issue) is

translated into an individual requirements behaviour tree (RBT). Each RBT has associated with it a so called “*precondition*” that needs to be satisfied by the system as a whole in order for the encapsulated behaviour to be applicable. This precondition is the root of the tree. It is either explicit in the requirements or implicit, in which case it has to be added when modelling the Behaviour Tree. (Note that adding implicit preconditions is a creative task that involves understanding of the problem and is not automatable.) We mark an added precondition with a ‘+’ (in its tag-frame). At least one other RBT has to establish this precondition and therefore provide a point of integration for the two trees. (Excluded from this rule is the precondition that becomes the root of the design tree as a whole.) As we integrate the RBTs, one at a time, we are constructing a model of the system design from its set of requirements.

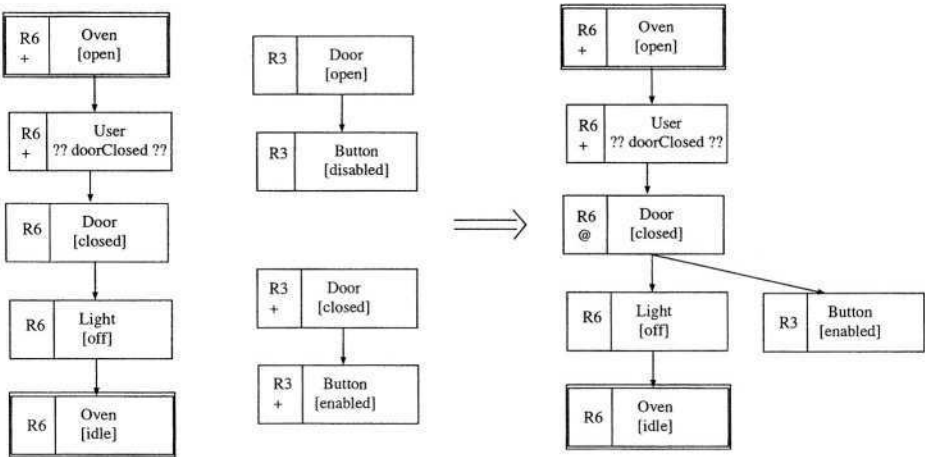


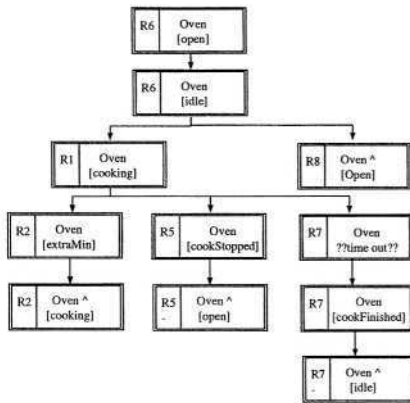
Fig. 4. Behaviour Trees of Requirements R6 and R3 and their integration

To demonstrate the approach we reproduce the example of the Microwave Oven as published in [Dro03].

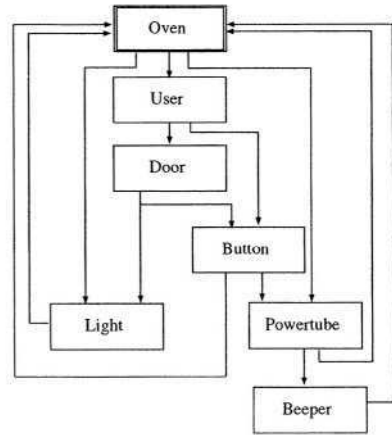
- R1. There is a single control button available for the user of the oven. If the oven is idle with the door closed and you push the button, the oven will start cooking (that is, energise the power-tube for one minute).
- R2. If the button is pushed while the oven is cooking it will cause the oven to cook for an extra minute.
- R3. Pushing the button when the door is open has no effect (because it is disabled).
- R4. Whenever the oven is cooking or the door is open the light in the oven will be on.
- R5. Opening the door stops the cooking.

- R6. Closing the door turns off the light. This is the normal idle state prior to cooking when the user has placed food in the oven.
- R7. If the oven times-out, the light and the power-tube are turned off and then a beeper emits a sound to indicate that the cooking is finished.

In order to demonstrate one integration step we show the RBTs for requirements R6 and R3 in Figure 4. Note the implicit preconditions in R6 (marked with a '+'): the oven must be open and the user has to close the door. Requirement R3 is also extended to model the behaviour of the button in case the door is closed. Two of the trees share a point of integration and can be grafted together. Note, that the point of integration, namely the box Door [closed], is marked with a '@' in the tag.



a.) Projected Behaviour of the Oven Component



b.) Architecture of interacting Components

Fig. 5. Different Views of the System Design

In a similar fashion all other individual RBTs are integrated into the tree. The result is called *Design Behaviour Tree* (DBT) (see Figure 6). Leaf nodes marked with a ^ symbol indicate a loop back to an earlier node in the tree. Note that requirement R8 was added to the tree after it was found missing in the original requirements.

By applying a filter to the DBT, one can extract the different component behaviours. We filter out all boxes that belong to a specific component. Figure 5a, for example, shows the behaviour of the oven component. Missing from this view, however, are the events that trigger the behaviour. The view is therefore incomplete.

An architectural view can be gained by applying a simple algorithm to the DBT, marking all components and interfaces between them (for more detail see [Dro03]). Figure 5b shows an architectural view of the oven system.

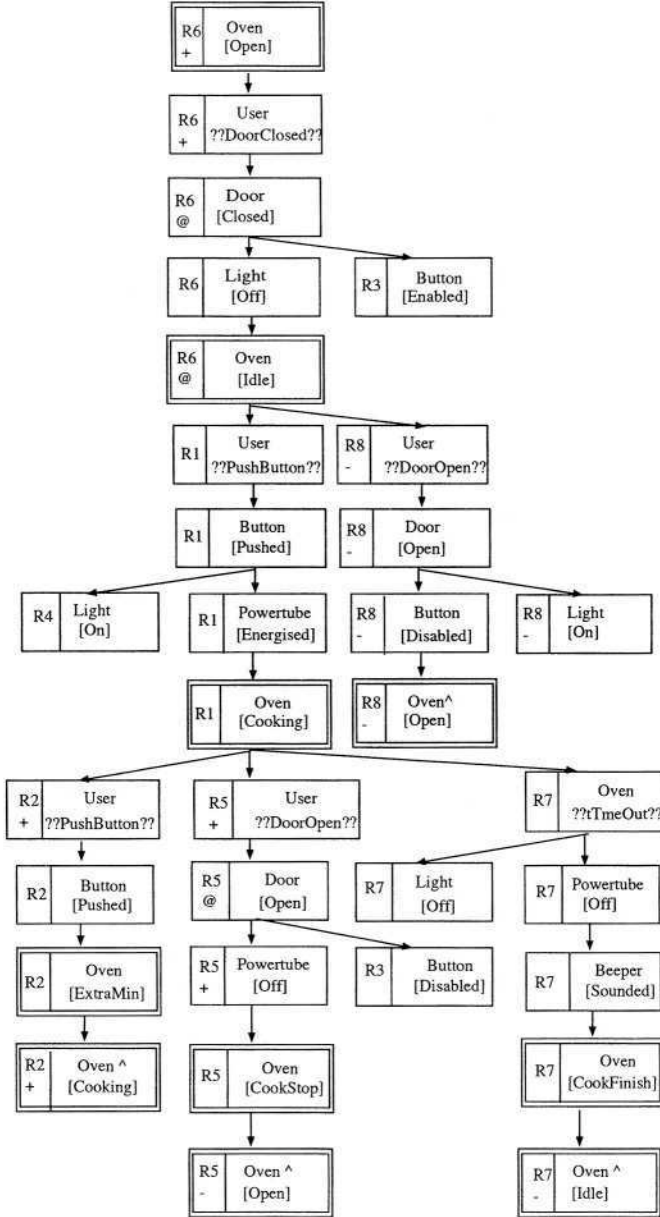


Fig. 6. Design Behaviour of the Microwave Oven

3 Integration of Behaviour Trees with CSP

We now introduce an integration of Behaviour Trees with CSP. By doing so, we provide the former notation with a formal semantics and the latter with a front-

end notation that supports a novel approach for modelling functional requirements. As shown in the previous section, Behaviour Trees provide a systematic and constructive way of capturing functional requirements in a system design model. Individual functional requirements are modelled as single Behaviour Trees in isolation and are later integrated into one Design Behaviour Tree. A similar stepwise approach could not easily be followed when using the CSP notation to model functional requirements. Points of integration for two individual requirements would be more difficult to determine. However given a Design Behaviour Tree that integrates the set of requirements, it is easy to see how this can be captured as interacting CSP processes. We first give a brief overview of the CSP notation as it is used in our approach.

3.1 The Notation of CSP

CSP (Communicating Sequential Processes) [Hoa85,Ros98] is a process algebra for modelling interacting components. Each component is specified through its behaviour which is given as a process. A process defines a sequence (or a set of sequences) of events that the process may undergo. This set of events is called the *alphabet* of a process. We model

$$P = a \rightarrow Q$$

to define that process P undertakes event a and then behaves like process Q .

Channels are a medium for transferring data and are used in a similar fashion as events. Output of data d on channel c is modelled as $c!d$, data input is modelled by $c?d$. Two processes synchronising on these two channel events perform a *handshake* communication and exchange the value of data d .

The external choice operator $\square$ provides a means to capture alternatives:

$$P = a \rightarrow Q \square b \rightarrow R$$

specifies that P does an a and then behaves like Q or does b and then continues like R depending on which event the environment of P is communicating, a or b .

Processes can run in parallel, $P \parallel Q$, in which case they have to synchronise on all events their alphabets have in common. It is possible to restrict the set of synchronising events by using the alphabetised parallel,

$$P \_A \parallel_B Q$$

where A and B are subsets of the alphabet of P and Q , respectively. In this case the processes P and Q synchronise on those events that the sets A and B have in common, i.e., the synchronisation set is given as $A \cap B$.

STOP, *SKIP* and *CHAOS*() are special processes. *STOP* models the unsuccessful termination of a process (like a deadlock), while *SKIP* represents the successful termination. The process *CHAOS*(A) models arbitrary behaviour

over the alphabet A . That is, the traces of this process are given as all possible sequences over events in set A .

A process's behaviour can also be guarded by a boolean expression over process parameters.

$$P = b \ \& \ Q$$

models that if b is true then P behaves like Q . Otherwise, if b is not true then P terminates unsuccessfully (i.e., equals *STOP*).

We also use the interrupt operator, Δ .

$$P = (a \rightarrow Q) \Delta (b \rightarrow R)$$

models that the process $a \rightarrow Q$ is interrupted if the event b occurs in which case P continues to behave like process R .

3.2 Translating a Behaviour Tree into CSP Processes

The semantics of a Behaviour Tree can be captured by interacting CSP processes. We translate the fully integrated Design Behaviour Tree (DBT) as a whole rather than the individual Requirements Behaviour Trees (RBTs). That is, we assume that the completion of individual trees (i.e., adding implicit preconditions etc.) and their integration into one single tree has already been done by the user.

Since Behaviour Tree Notation is not (yet) equipped with a formal semantics our translation is described in an algorithmic fashion rather than being fully formalised. Note that we are aiming at an automatable translation process.

In the following, we describe our translation procedure mostly in terms of the given example of the Microwave Oven in order to illustrate the process. This, however, does not limit the applicability of our approach to this example. In cases where features of the notation are not contained in the oven example, we introduce abstract examples for illustration.

Generally, each component in the DBT is modelled as a CSP component with its behaviour defined as a process. These CSP components run in parallel and have to synchronise on all events they have in common.

A component process is divided into sub-processes. Each sub-process reflects a state change that the component exhibits between the appearance of two of its boxes in the DBT. Usually, a state change is triggered by an event box that appears between two boxes of the component. In order to determine the sub-processes for each component, we have to traverse each branch in the tree.

The name of sub-process and events are derived from the component name, the state name and the event name respectively, as they are given in the Behaviour Tree. We follow the CSP convention that process names are capitalised whereas event names are not.

Given the Design Behaviour Tree of the Microwave Oven in Figure 6, for example, we traverse the tree to define a sub-process for each state realisation box, e.g., for the box *Oven* [Open] we define *OvenOpen* as a sub-process of component *Oven*, for the box *Door* [Closed] we define *DoorClosed* as a sub-process

of component *Door*. In addition, we define an initial sub-process for each component other than the system component. This sub-process starts at the root node.

We might start to naturally translate the DBT into the following sub-process for the *Oven* component:

$$OvenOpen = userDoorClosed \rightarrow OvenIdle$$

The initial sub-processes for the components *Door* and *Light* are

$$\begin{aligned} DoorInit &= userDoorClosed \rightarrow DoorClosed \\ LightInit &= userDoorClosed \rightarrow LightOff \end{aligned}$$

The CSP components of the system, like *Oven*, *Light* and *Door*, are running in parallel and have to synchronise on the events in common, e.g., *userDoorClosed*. This synchronisation on events that occur in the DBT, however, does not guarantee that the components get control in the right order. The three sub-processes above, when running in parallel, will change concurrently the state of all components, the *Oven*, the *Light* and the *Door*. Even if in this case study this might be acceptable, in general it is not.

To overcome this problem, we augment the edges in the tree with additional events e_i as shown in Figure 9. Branching edges that model concurrent state realisation share the same event (e.g., two edges are labelled by event e_3). A single outgoing edge from two concurrent state realisations is duplicated so that each box has an outgoing edge. Both edges carry the same label (e.g., two edges are labelled with e_4). In case of a selected flow, selected event, and threaded control flow, each edge is labelled individually (e.g., edges labelled by events e_{13} , e_{17} , and e_{22}). These additional events ensure that the state changes of the components, when running in parallel, happen in the same order as indicated in the DBT. Whenever a component gives control to the component in the next box this is marked through an event, namely the event that labels the outgoing edge. Similarly, whenever a component gets control this is marked by the event that labels its ingoing edge.

Each sub-process now describes the control flow in the tree up to the next box of the same component in terms of the events along the edges and the DBT events. We define our three sub-processes from above as follows:

$$\begin{aligned} DoorOpen &= e_1 \rightarrow userDoorClosed \rightarrow e_2 \rightarrow DoorClosed \\ LightOn &= e_1 \rightarrow userDoorClosed \rightarrow e_2 \rightarrow e_3 \rightarrow LightOff \\ OvenOpen &= e_1 \rightarrow userDoorClosed \rightarrow e_2 \rightarrow e_3 \rightarrow e_4 \rightarrow OvenIdle \end{aligned}$$

All three sub-processes synchronise on the events $\{e_1, userDoorClosed, e_2\}$, and *LightOn* and *OvenOpen* will also synchronise on event e_3 .

Generally, the processes have to interact on all events their individual alphabets have in common. Process internal events (that do not contribute to the synchronisation between processes) are only those events that are not used by

any other process. In an augmented Behaviour Tree these internal events label edges between two boxes that belong to the same component (e.g., e_{21} in Figure 9).

To simplify the CSP processes, we aim to minimise the number of events involved in the processes. We observe that each sub-process has to synchronise only on those events that determine when control is passed from itself onto another component and when control is passed back to itself and a state change will occur. Additionally, we want to keep track of the DBT events. In principle, each sub-process needs to synchronise on three events:

1. the event labelling the *outgoing* edge of the box that corresponds to the sub-process;
example: *OvenOpen* has to synchronise on e_1
2. the DBT event that *triggers* the state change;
example: *OvenOpen* has to synchronise on *userDoorOpen*
3. the event labelling the *incoming* edge of the next box of the component marking the follow-on sub-process;
example: *OvenOpen* has to synchronise on e_4 .

Moreover, DBT events can be identified with the events that label their ingoing and outgoing edges. Since the event boxes are not translated into sub-processes, we only need one event here instead of three. For instance, the sequence $e_1 \rightarrow \textit{userDoorClosed} \rightarrow e_2$ simplifies to *userDoorClosed*. However, we have to distinguish between multiple occurrences of the same event in the tree. Therefore, we number the DBT events if necessary (e.g., *userDoorOpen*₁ and *userDoorOpen*₂ as indicated in Figure 9).

According to these simplifications, the sub-processes reduce to

$$\begin{aligned}\textit{DoorOpen} &= \textit{userDoorClosed} \rightarrow \textit{DoorClosed} \\ \textit{LightOn} &= \textit{userDoorClosed} \rightarrow e_3 \rightarrow \textit{LightOff} \\ \textit{OvenOpen} &= \textit{userDoorClosed} \rightarrow e_4 \rightarrow \textit{OvenIdle}.\end{aligned}$$

It becomes more apparent how the synchronisation works if we consider the follow-on sub-processes for the *Door* and the *Light* component:

$$\begin{aligned}\textit{DoorClosed} &= e_3 \rightarrow \textit{userDoorOpen} \rightarrow \dots \\ \textit{LightOff} &= e_4 \rightarrow \dots\end{aligned}$$

By synchronising on event e_3 , we ensure that *LightOff* can only be reached once *DoorClosed* has started. Similarly, the synchronisation on e_4 guarantees that *OvenIdle* can only happen after *LightOff* has started. This corresponds to the sequence of boxes in the tree.

Data flow boxes for input and output as shown in Figure 1(e) and 1(f) are modelled with CSP channels. We introduce a channel c_i for each pair of data flow boxes, assuming that these always follow each other in the Behaviour Tree. The two components that are involved in the data exchange synchronise in a handshake fashion on the CSP events $c_i!data$ and $c_i?data$.

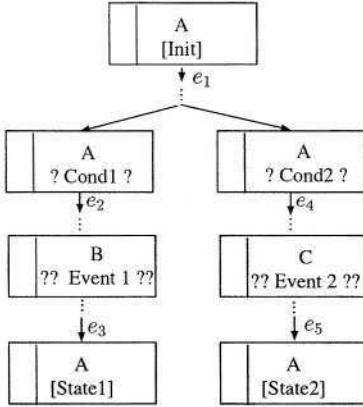


Fig. 7. Selected Flow of Control

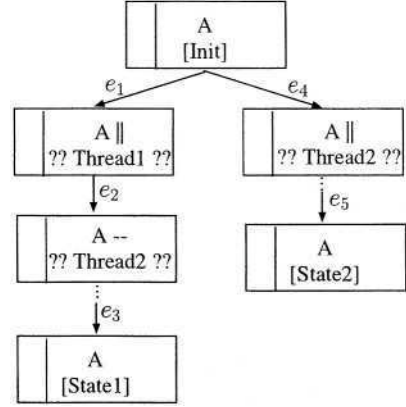


Fig. 8. Behaviour Tree with Threads

3.3 Translating Modes of Control Flow

The procedure described above captures our translation into CSP for sequential flow of control. It also subsumes modelling ‘concurrent flow’ (as depicted in Figure 3(b)). Concurrent flow in a Behaviour Tree denotes a state change of two components happening at the same time. We capture this kind of concurrency in our CSP model by running all corresponding CSP components in parallel.

Other modes of the control flow of Behaviour Trees are selected event, selected flow and threads (see Figure 3(c), (d) and (e)). A selected event branch is modelled by means of the external choice operator: depending on the event provided by the environment one branch of the sub-process will be chosen. For example, given the DBT in Figure 9 the sub-process *OvenCooking* is modelled as follows:

$$\begin{aligned}
 \text{OvenCooking} &= \text{userPushButton} \rightarrow e_{15} \rightarrow \text{OvenExtraMin} \\
 &\quad \square \text{userDoorOpen} \rightarrow e_{20} \rightarrow \text{OvenCookStopped} \\
 &\quad \square \text{ovenTimeout} \rightarrow e_{25} \rightarrow \text{OvenCookFinished}
 \end{aligned}$$

Selected flow in a Behaviour Tree can be modelled utilising a combination of guarded event and external choice operator. Usually conditions are not public to all components since their truth value depends on the attributes (i.e., parameters) of a particular component and has to be decided locally. In the tree depicted in Figure 7, the control branches depend on condition *Cond1* or *Cond2* being satisfied in component *A*. (The ... in the tree indicate that more boxes might stand between the boxes of component *A* and are omitted here.) We translate this scenario into the following CSP sub-process for component *A*:

$$\begin{aligned}
 AInit &= e_1 \rightarrow (aCond_1 \ \& \ (e_2 \rightarrow bEvent_1 \rightarrow e_3 \rightarrow Astate1) \\
 &\quad \square \ aCond_2 \ \& \ (e_4 \rightarrow cEvent_2 \rightarrow e_5 \rightarrow Astate2))
 \end{aligned}$$

The choice between the two branches is guarded: if $aCond_1$ is true the process *AInit* behaves like $bEvent_1 \rightarrow e_2 \rightarrow Astate1$. Otherwise, if $aCond_1$ is not true

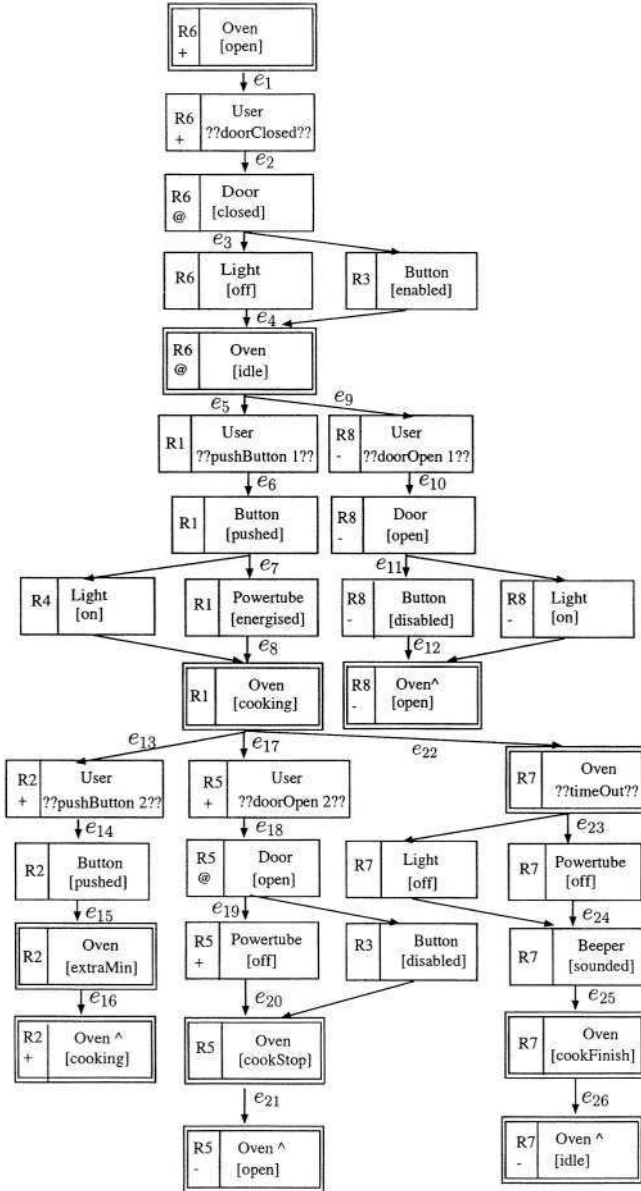


Fig. 9. Augmented Design Behaviour Tree of the Microwave Oven

this branch does not terminate successfully, it behaves like *STOP*. The second branch describes similar behaviour depending on the truth of $aCond_2$. If one of the choices cannot terminate successfully because the guard is not satisfied the choice operator will choose the other branch of the choice. Other components of the system are usually not able to decide on the truth of conditions that depend

on the state of one component. However, due to our synchronisation mechanism they are forced to follow the selected flow in correspondence to the component that is responsible for the selection, which is component *A* in the given case.

Concurrent control flow and threads are captured similarly by the CSP parallel operator combining the branches of the sub-tree in the sub-processes. To kill a thread we utilise the interrupt operator. We give an abstract example in Figure 8. The component *A* starts with its initial state *Init*. After that the behaviour branches into two threads triggered by the two events *Thread1* and *Thread2*. The occurrence of each of these events starts a new individual process, a thread. In this example, the thread in the left branch kills the thread in the right branch as depicted by the *A--??Thread2??* box.

We model this Behaviour Tree in CSP by the following process:

$$\begin{aligned} AInit &= aThread1 \rightarrow killAThread2 \rightarrow e_3 \rightarrow AState1 \\ &\parallel (aThread2 \rightarrow e_5 \rightarrow AState2) \Delta (killAThread2 \rightarrow STOP) \end{aligned}$$

This process has two sub-processes that run in parallel. The first one is triggered by event *aThread1*, the second one by *aThread2*. We introduce a kill-event for the corresponding box, namely *killAThread2*. This kill-event activates the interrupt that is modelled in the second sub-process. As soon as it occurs the sub-process (*aThread2* $\rightarrow$ *e*₅ $\rightarrow$ *AState2*) will be interrupted and terminates due to the process *STOP*. Note that the additional labels *e*₁, *e*₂, and *e*₄ in our abstract example above are not used in the CSP model since they are merged with the given DBT events.

4 Example

In this section, we give the full view of the CSP model of the Microwave Oven. For the translation we took the Design Behaviour Tree (DBT) augmented with additional events as shown in Figure 9. The modelling follows the description given in Section 3.

The translation of the DBT results in the following CSP model. Traversing the tree we get a set of sub-processes for the components involved.

The *Oven* component comprises six sub-processes as the Behaviour Tree shows six state realisation boxes for this component.

$$\begin{aligned} OvenOpen &= userDoorClosed \rightarrow e_4 \rightarrow OvenIdle \\ OvenIdle &= userPushButton_1 \rightarrow e_8 \rightarrow OvenCooking \\ &\quad \square userDoorOpen_1 \rightarrow e_{12} \rightarrow OvenOpen \\ OvenCooking &= userPushButton_2 \rightarrow e_{15} \rightarrow OvenExtraMin \\ &\quad \square userDoorOpen_2 \rightarrow e_{20} \rightarrow OvenCookStop \\ &\quad \square ovenTimeOut \rightarrow e_{25} \rightarrow OvenCookFinish \\ OvenExtraMin &= e_{16} \rightarrow OvenCooking \\ OvenCookStop &= e_{21} \rightarrow OvenOpen \\ OvenCookFinish &= e_{26} \rightarrow OvenIdle \end{aligned}$$

At the leaves of the tree the branches loop back to the boxes *Oven[Cooking]*, *Oven[Open]* and *Oven[Idle]*, respectively. Accordingly, the sub-processes *OvenExtraMin*, *OvenCookStopped* and *OvenCookFinished* loop back to the earlier sub-processes. Note that we distinguish the two occurrences of events *userDoorOpen* and *userPushButton* through indexes. Similarly, we get the following sub-processes for the components *Door* and *Light*.

$$\begin{aligned}
& \text{DoorInit} = \text{userDoorClosed} \rightarrow \text{DoorClosed} \\
& \text{DoorClosed} = e_3 \rightarrow (\text{userDoorOpen}_1 \rightarrow \text{DoorOpen} \\
& \quad \square \text{userDoorOpen}_2 \rightarrow \text{DoorOpen}) \\
& \text{DoorOpen} = e_{19} \rightarrow \text{userDoorClosed} \rightarrow \text{DoorClosed} \\
& \quad \square e_{11} \rightarrow \text{userDoorClosed} \rightarrow \text{DoorClosed} \\
& \text{LightInit} = \text{userDoorClosed} \rightarrow e_3 \rightarrow \text{LightOff} \\
& \text{LightOff} = e_4 \rightarrow (\text{userPushButton}_1 \rightarrow e_7 \rightarrow \text{LightOn} \\
& \quad \square \text{userDoorOpen}_1 \rightarrow e_{11} \rightarrow \text{LightOn}) \\
& \quad e_{24} \rightarrow (\text{userPushButton}_1 \rightarrow e_7 \rightarrow \text{LightOn} \\
& \quad \square \text{userDoorOpen}_1 \rightarrow e_{11} \rightarrow \text{LightOn}) \\
& \text{LightOn} = e_8 \rightarrow (\text{ovenTimeOut} \rightarrow \text{LightOff} \\
& \quad \square \text{LightInit}) \\
& \quad \square e_{12} \rightarrow \text{userDoorClosed} \rightarrow e_3 \rightarrow \text{LightOff}
\end{aligned}$$

When translating the Design Behaviour Tree into sub-processes of the CSP components, we have to follow each branch of the tree for each component. For instance, although the *Light* component is not involved in the branches following label event e_{13} and e_{17} we have to cater for these as a possible behaviour of the overall system with which the *Light* component has to synchronise. This results in an additional choice for the *LightOn* process, namely $e_8 \rightarrow \text{LightInit}$. The behaviour of component *Button* is defined through the following sub-processes.

$$\begin{aligned}
& \text{ButtonInit} = \text{userDoorClosed} \rightarrow e_3 \rightarrow \text{ButtonEnabled} \\
& \text{ButtonEnabled} = e_4 \rightarrow (\text{userDoorOpen}_1 \rightarrow e_{11} \rightarrow \text{ButtonDisabled} \\
& \quad \square \text{userPushButton}_1 \rightarrow \text{ButtonPushed}) \\
& \text{ButtonDisabled} = e_{12} \rightarrow \text{ButtonInit} \\
& \quad \square e_{20} \rightarrow \text{ButtonInit} \\
& \text{ButtonPushed} = e_7 \rightarrow (\text{userPushButton}_2 \rightarrow \text{ButtonPushed} \\
& \quad \square \text{userDoorOpen}_2 \rightarrow e_{19} \rightarrow \text{ButtonDisabled} \\
& \quad \square (\text{userDoorOpen}_1 \rightarrow e_{11} \rightarrow \text{ButtonDisabled} \\
& \quad \quad \square \text{userPushButton}_1 \rightarrow \text{ButtonPushed})) \\
& \quad \square e_{15} \rightarrow (\text{userPushButton}_2 \rightarrow \text{ButtonPushed} \\
& \quad \quad \square \text{userDoorOpen}_2 \rightarrow e_{19} \rightarrow \text{ButtonDisabled} \\
& \quad \quad \square (\text{userDoorOpen}_1 \rightarrow e_{11} \rightarrow \text{ButtonDisabled} \\
& \quad \quad \quad \square \text{userPushButton}_1 \rightarrow \text{ButtonPushed}))
\end{aligned}$$

Similarly to the sub-process *LightOn* above, the process *ButtonPushed* has additional choices after synchronising on events e_7 and e_{15} . In both cases, the overall system will reach the selected event branches and may choose to synchronise on event e_{22} next. The button component is not apparent in this branch.

However, it has to synchronise on the events that follow the loop-back point $Oven[Idle]$. This results in the additional choices ($userDoorOpen_1 \rightarrow e_{11} \rightarrow ButtonDisabled \sqcap userPushButton_1 \rightarrow ButtonPushed$) after events e_7 and e_{15} .

The sub-processes of components *Powertube* and *Beeper* are not affected by branches of the tree to which they do not contribute. Consider, for example, component *Powertube*: one of these branches is starting with event e_9 and loops back to the root state $Oven[Open]$. At this point the *Powertube* is still in sub-process *PowertubeInit* and waits for the first *userPushButton* event. The traversing of this branch does not lead to an additional choice in sub-process *PowertubeInit*. A similar observation can be made for each branch the components do not contribute.

The translation for components *Powertube* and *Beeper* therefore results in fairly simple sub-processes as shown below.

$$\begin{aligned} PowertubeInit &= userPushButton_1 \rightarrow e_7 \rightarrow PowertubeEnergised \\ PowertubeEnergised &= e_8 \rightarrow (userDoorOpen_2 \rightarrow e_{19} \rightarrow PowertubeOff \\ &\quad \sqcap ovenTimeOut \rightarrow PowertubeOff) \\ PowertubeOff &= e_{20} \rightarrow PowertubeInit \\ &\quad \sqcap e_{24} \rightarrow PowertubeInit \end{aligned}$$

$$\begin{aligned} BeeperInit &= ovenTimeOut \rightarrow e_{24} \rightarrow BeeperSounded \\ BeeperSounded &= e_{25} \rightarrow BeeperInit \end{aligned}$$

The components are defined as being equal to the initial sub-processes, i.e., those starting at the root node of the DBT.

$$\begin{aligned} Oven &= OvenOpen \\ Door &= DoorInit \\ Light &= LightInit \\ Button &= ButtonInit \\ Powertube &= PowertubeInit \\ Beeper &= BeeperInit \end{aligned}$$

In order to define the parallel composition of the components, we define the alphabets of each of them. According to the reduced number of events, the alphabets are reduced to a subset of the overall event alphabet as apparent in the augmented tree. The single alphabets are listed as follows:

- alphabet of *Oven*: $\alpha = \{e_4, e_8, e_{12}, e_{15}, e_{16}, e_{20}, e_{21}, e_{25}, e_{26}, userDoorClosed, userDoorOpen_1, userDoorOpen_2, userPushButton_1, userDoorOpen_2, ovenTimeOut\}$
- alphabet of *Door*: $\beta = \{e_3, e_{11}, e_{19}, userDoorClosed, userDoorOpen_1, userDoorOpen_2\}$
- alphabet of *Light*: $\gamma = \{e_3, e_4, e_7, e_8, e_{11}, e_{12}, e_{24}, userDoorOpen_1, userDoorClosed, userPushButton_1, ovenTimeOut\}$
- alphabet of *Button*: $\delta = \{e_3, e_4, e_{11}, e_{12}, e_{15}, e_{19}, e_{20}, userDoorOpen_1, userDoorOpen_2, userDoorClosed, userPushButton_1, userPushButton_2\}$

- alphabet of *Powertube*: $\varepsilon = \{e_8, e_7, e_{19}, e_{20}, e_{24}, \text{userDoorOpen}_2, \text{userPushButton}_1, \text{ovenTimeOut}\}$
- alphabet of *Beeper*: $\zeta = \{e_{24}, e_{25}, \text{ovenTimeOut}\}$

The alphabet of the overall system is the union of the alphabets of all components, i.e.,

$$\begin{aligned} \Sigma &= \alpha \cup \beta \cup \gamma \cup \delta \cup \varepsilon \cup \zeta \\ &= \{e_3, e_4, e_7, e_8, e_{11}, e_{12}, e_{15}, e_{16}, e_{19}, e_{20}, e_{21}, e_{24}, e_{25}, e_{26}, \\ &\quad \text{userDoorClosed}, \text{userDoorOpen}_1, \text{userDoorOpen}_2, \\ &\quad \text{userPushButton}_1, \text{userPushButton}_2, \text{ovenTimeOut}\} \end{aligned}$$

The system is now defined as the parallel composition of all components where each component synchronises over its own alphabet:

System =

$$\text{Oven } \alpha \parallel_{\Sigma_1} (\text{Door } \beta \parallel_{\Sigma_2} (\text{Light } \gamma \parallel_{\Sigma_3} (\text{Button } \delta \parallel_{\Sigma_4} (\text{Powertube } \varepsilon \parallel_{\zeta} \text{Beeper}))))$$

where

$$\begin{aligned} \Sigma_1 &= \beta \cup \gamma \cup \delta \cup \varepsilon \cup \zeta \\ \Sigma_2 &= \gamma \cup \delta \cup \varepsilon \cup \zeta \\ \Sigma_3 &= \delta \cup \varepsilon \cup \zeta \\ \Sigma_4 &= \varepsilon \cup \zeta \end{aligned}$$

The simplification through the reduced number of events that the components need to communicate reduces the size of the model substantially and thus helps to improve the efficiency of the analysis step.

Unlike the projected behaviour from the DBT (as shown, for example, in Figure 5a), the view of a single component in the CSP model is complete in terms of the DBT events that trigger the behaviour of the component. This component model may guide the further development of the system components.

5 Analysis of the CSP Model

For analysing the model we use the model checker FDR (Failure Divergence Refinement) [For96]. FDR supports checking deadlock, livelock and determinism of single CSP processes and allows checking the refinement relations (trace, failure, and failure divergence) between two CSP processes.

For example, we utilise FDR to check if our model, which is constructed from functional requirements, satisfies safety properties of the system. Safety properties are not necessarily stated as requirements in the requirements document so that it seems useful to check if they are satisfied by the model of the given requirements. Incompleteness and inconsistencies of the functional requirements will show through a violation of the safety properties.

One safety property for the Microwave Oven that we might want to check is:

The power-tube should not be energised when the door is open.

We model this property as a CSP process using the events for opening and closing the door and for pushing the button as they were used in the system. We define the set

$$\vartheta = \{userDoorOpen_1, userDoorOpen_2, userDoorClosed, \\ userPushButton_1, userPushButton_2\}.$$

The last two events are responsible for starting the power-tube. The user may push the button arbitrarily often but as soon as the door is opened, it has to be closed again before the two *userPushButton* events are available again. This can be modelled by the processes *Q* and *P* below. The process *Safety* is then defined as behaving like *P* on the events in ϑ . The behaviour on all other events (defined through set *Others*) is unrestricted (modelled as *Chaos*(*Others*)).

$$\begin{aligned} Others &= diff(\Sigma, \vartheta) \\ Q &= userPushButton_1 \rightarrow Q \\ &\quad \square userPushButton_2 \rightarrow Q \\ &\quad \square userDoorOpen_1 \rightarrow P \\ &\quad \square userDoorOpen_2 \rightarrow P \\ P &= userDoorClosed \rightarrow Q \\ Safety &= P \parallel\parallel CHAOS(Others) \end{aligned}$$

We checked trace refinement between the process *Safety* and the system and no violation was found. Due to the fact that the given example is very small the model checking process terminated after very short time.

Several deadlock checks on single CSP components and on the system as a whole were executed in order to debug our (so far hand-translated) CSP model. Here we found it very useful to read the counter-examples that are output by the FDR tool with the help of the given Design Behaviour Tree. The sequence of events in the counter-example showed which branch in the tree the control flow had taken. Generally, the given Design Behaviour Tree can be utilised to visualise the counter-examples in cases where a deadlock occurs or a safety property is violated.

6 Conclusion

We described the integration of Behaviour Trees and CSP. Behaviour Trees is a graphical notation for requirements engineering. The user models each individual functional requirement in isolation. The resulting individual requirements trees are later on integrated into a single tree. A Behaviour Tree takes a view on all components involved in the systems behaviour. This allows the user to translate textual requirements quite easily into this notation. We model this multi-component behaviour by means of communicating CSP processes. Each process is captured in terms of sub-processes which model the state changes of that component. In order to model the sequence of state changes of different

components, we augment the edges of the tree with additional events. The CSP components synchronise on these events as well as on the events that are given in the tree. We intend to exploit the model checker FDR for the analysis of the requirements model. To optimise the model we minimised the events that are involved in the synchronisation: each component refers only to the events labelling the edge outgoing from a state and the edge in-going to the follow-on state as well as the event in the tree which triggers the state change. This optimisation reduced the size of the CSP model significantly.

We used the model checker FDR for the analysis of the requirements model. We additionally modelled safety properties of the given system as a CSP process and checked if the requirements model satisfies those by utilising the refinement relation between the two models.

Our approach provides a formal semantics for parts of the notation of Behaviour Trees, and with this tool support for analysis. It also supports the user with a graphical representation for a subset of the CSP language for ease of communication with customers. This becomes apparent when the Behaviour Tree can be utilised for visualising the output of the FDR tool: the sequence of events in a counter-example shows the branch in the tree that represents the particular trace. Moreover, the Behaviour Tree approach provides the CSP user with support for requirements engineering.

The work in this paper handles only a sub-set of the Behaviour Tree Notation. Future work will deal with unresolved issues of remaining language constructs. These involve specifically the notation for data structures provided by the Behaviour Tree Notation.

Acknowledgements. I acknowledge the support of Australian Research Council (ARC) Discovery Grant DP0345355, *Building Dependability into Complex, Computer-based Systems*. I would also like to thank Ian Hayes, David Carrington, Peter Lindsay and Geoff Dromey for fruitful discussions and inspiration for this work. Thanks also to Graeme Smith and the anonymous reviewers whose comments helped to improve the draft of this paper.

References

- [BD00] C. Bolton and J. Davies. Activity graphs and processes. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *Int. Conference on Integrated Formal Methods (IFM 2000)*, volume 1945 of *Lecture Notes in Computer Science*, pages 77 – 96. Springer-Verlag, 2000.
- [BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modelling Language User Guide*. Addison-Wesley, 1999.
- [Dro03] R.G. Dromey. From requirements to design: Formalizing the key steps. In A. Cerone and P. Lindsay, editors, *Int. Conference on Software Engineering and Formal Methods (SEFM 2003)*, pages 2–11. IEEE Computer Society, 2003.
- [For96] Formal Systems (Europe) Ltd. *Failure Divergence Refinement, FDR 2.0, User Manual*, August 1996.

- [Har87] D. Harel. Statecharts: Visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
- [HD99] D. Harel and W. Damm. LSCs: Breathing life into message sequence charts. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 99)*, pages 293 – 312. Kluwer Academic Publishers, 1999.
- [Hoa85] C.A.R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice Hall, 1985.
- [NB02] M.Y. Ng and M. Butler. Tool support for visualizing CSP in UML. In C. George and H. Miao, editors, *Int. Conference on Formal Engineering Methods (ICFEM 2002)*, volume 2495 of *Lecture Notes in Computer Science*, pages 287 – 298. Springer-Verlag, 2002.
- [Ros98] A.W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.

Generating MSCs from an Integrated Formal Specification Language

Jin Song Dong¹, Shengchao Qin², and Jun Sun^{1*}

¹ School of Computing
National U. of Singapore
{dongjs,sunj}@comp.nus.eud.sg
² Singapore-MIT Alliance
National U. of Singapore
qinsc@comp.nus.edu.sg

Abstract. The requirements capture of complex systems requires powerful mechanisms for specifying system state, structure and interactive behaviors. Integrated formal specification languages are well suited for presenting more complete and coherent requirement models for complex systems. Given an integrated model, one can project it into multiple views for specialized analysis. Message Sequence Charts (MSCs) is a popular graphical notation for presenting interactive viewpoints of a system. In this paper, we investigate the semantic based transformation from an integrated formal specification language TCOZ to MSCs. An automated tool has also been developed for generating MSCs from TCOZ models. Furthermore, by inserting operation constraints (as assertions) into the generated MSCs, system testing requirements can be obtained.

Keywords: Requirement Engineering, TCOZ, MSC

1 Introduction

Multi-viewpoints [3,21] are effective techniques for capturing complex system requirements. Various formal notations are often used for presenting different viewpoints for large and complex systems which may have intricate system states and complex concurrent and interactive behaviors. The formal link and consistency issues between the viewpoint models represented in different formalisms remain as a challenging research topic. Recent investigations on linkings between different formalisms [5,13,17,26,29,30,32] may provide some meta support to the issues of viewpoints consistency. One such linked formalism is Timed Communicating Object Z (TCOZ) [17] which builds on the strengths of Object-Z [11,25] in modeling complex data and state with the strengths of TCSP [9] in modeling process control and real-time interactions. TCOZ can be well suited for presenting more complete and coherent requirement models that comprehend various

* Author for correspondence, fax: +65 6779 4580, phone: +65 6874 4353

viewpoints for complex systems. Given an integrated model, one can project it into consistent multiple views for specialized analysis. In this paper, we are interested in one particular viewpoint projection – the communication and interaction perspective. Message Sequence Charts (MSCs) [15] is a popular graphical notation for presenting interactive viewpoints of a system. In this paper, we investigate the semantic based transformation from TCOZ (trace models) to MSCs (process models). By identifying a set of traces with MSCs, the cause and effect relations between partial ordering events in concurrent systems [10] can be captured. An automated tool has also been developed in Java for generating MSCs from TCOZ models. Furthermore, by inserting class invariants and operation constraints (as assertions) into the generated MSCs (execution scenarios), system testing requirements can be obtained.

Various attempts to combine formal specifications with graphical notations have been explored [2,6,7,8,16,20,22]. Bolton and Davies [2] has given a process semantics in CSP for UML activity diagrams. They use the process semantics to demonstrate the consistency of the object model. Our approach is to automatically generate MSCs based on its standard process semantics. Brooke and Paige [4] have recently developed a tool-supported graphical notation for TCSP. The difference between Brooke and Paige's approach and ours is that we use existing formal graphical notations instead of creating new ones. Ng and Butler [20] have developed a tool for visualizing CSP in UML for both the static architecture and the dynamic behaviors. In our approach, we are particularly interested in capturing dynamic interactions between objects.

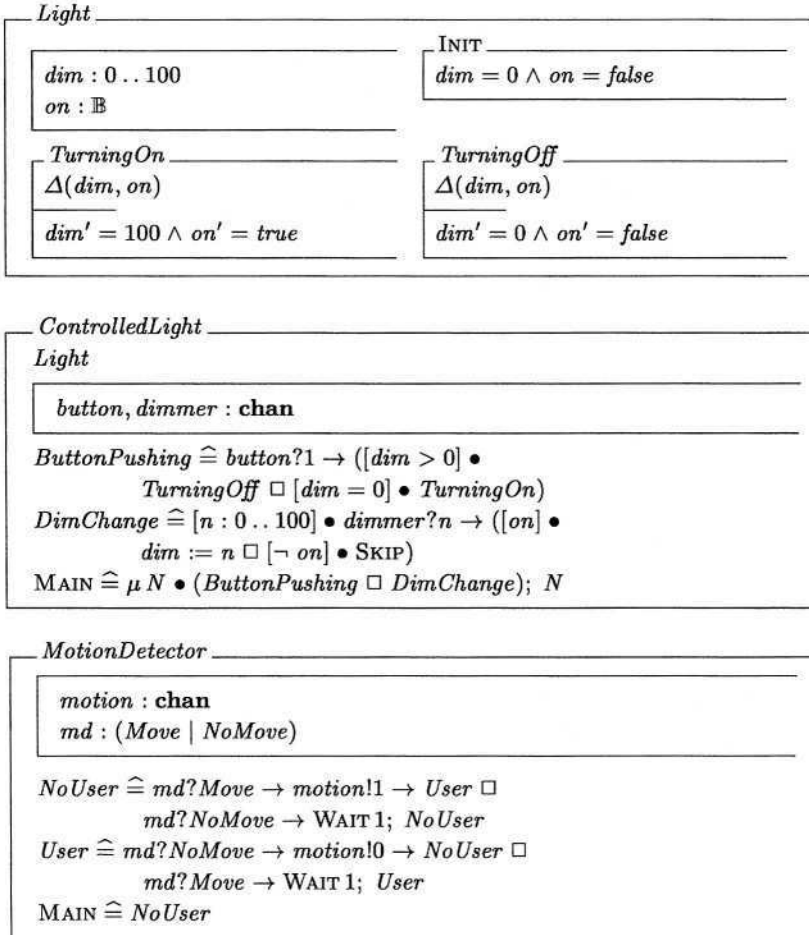
The remainder of the paper is organized as follows. Section 2 briefly introduces the technical background, the TCOZ notation and MSCs, both basic MSCs (BMSCs) and high-level MSCs (HMSCs). Section 3 presents the link between TCOZ and MSCs, both BMSCs and HMSCs, explains how to generate system test requirements from TCOZ specifications. Section 4 presents an XML-based automatic transformation tool built using Java. Section 5 concludes the paper.

2 Overview of TCOZ and MSCs

2.1 Overview of TCOZ

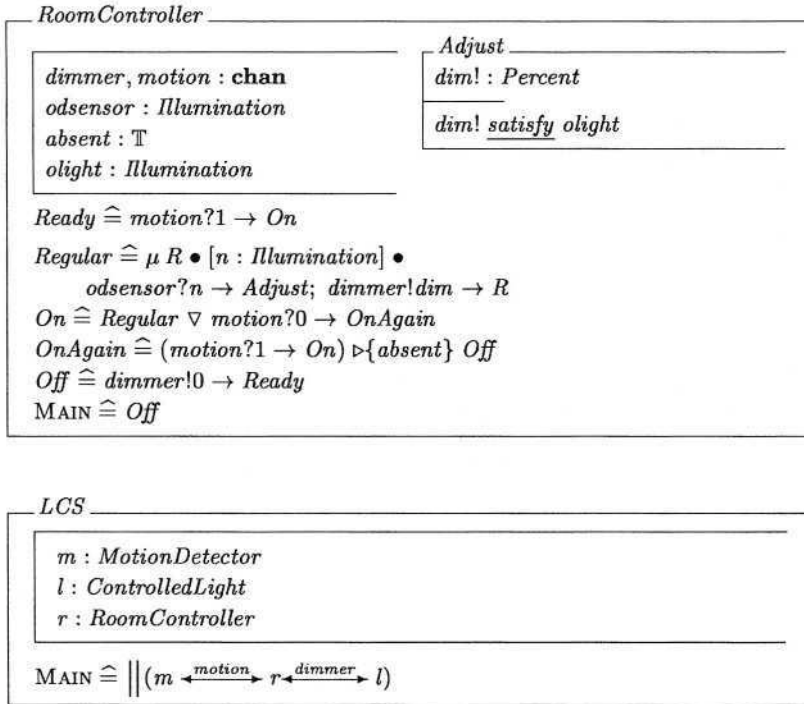
TCOZ is a blending of Object-Z and TCSP. The basic structure of a TCOZ document is the same as for Object-Z, consisting of a sequence of definitions, including type and constant definitions in the usual Z style. TCOZ varies from Object-Z in the structure of class definitions, which may include CSP channel and processes definitions. Channels in TCOZ are defined as communication interfaces between objects. All dynamic interactions between objects must take place through channel communication mechanism. The true power of TCOZ comes from the ability to make use of TCSP primitives in describing the process aspects of an operation's behavior. All operation definitions in TCOZ are TCSP process definitions, with operation schemas identified with Timed CSP processes. The data-related aspects of TCOZ are modeled using state bindings and the process-related aspects are modeled using event traces and refusals [18].

We take a simplified version of the Light Control System (LCS) [12] to illustrate the features of TCOZ and as an example to demonstrate the projection from TCOZ to MSCs. LCS is an intelligent control system. It can detect the occupation of a building, then turn on or turn off the lights automatically. It is able to tune illumination (in percentage) in the building according to the outside light level. A typical system behavior is that when a user enters a room: a motion detector senses the presence of the person, the room controller reacts by receiving the current daylight level and turning on the light group with appropriate illumination setting (let *satisfy* represent the relationship between daylight level and required illumination). When a user leaves a room (leaving it empty): the detector senses no movement, the room controller waits for *absent* time units and then turns off the light group. The occupant can directly turn on/off the light by pushing the button.



Class *Light* is essentially an Object-Z class (passive class). Class *ControlledLight*, a subclass of *Light*, extends *Light* class with process definitions, *ButtonPushing*, *DimChange* and *MAIN*. A *MAIN* process indicates that *ControlledLight* defines an active object, which has its own thread of control. It is used to determine the behavior of objects of an active class after initialization. *button* and *dimmer* are defined as channels connecting the light to the environment and room controller.

A motion detector detects any movement in the room so to tell whether some one is in and send proper signal on channel *motion*. $\square$ denotes a choice made by the environment.



A room controller communicates with the motion detector and light by declaring channels with same names as those in the respective classes. It takes in signals from the motion detector and sends proper signal to the light. Its behavior is captured by interrupt ∇ and timeout $\triangleright$ operators. Finally, a light control system is composed by a room controller, a motion detector and a light.

2.2 Overview of MSCs Language

The language MSCs is standardized by International Telecommunication Union (ITU). It provides a mean for visualization of the interaction of system components, either graphically or textually. The core of MSCs is called Basic Message

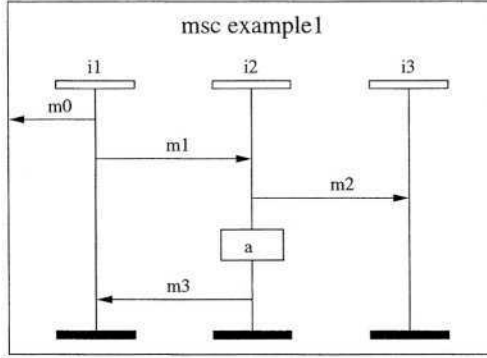


Fig. 1. A BMSC Example

Sequence Charts (BMSCs), which concerns communications and actions only. Then, additional basic concepts like process creation, termination, time handling, incomplete message events and conditions are added. Later, more complicated constructs are introduced. They are inline expressions, MSC reference expressions and High-level Message Sequence Charts (HMSCs), which enrich MSCs with intricate possibilities of describing complex systems.

A simple example of BMSCs is given as Figure 1. Each vertical line represents an active component (Z.120 terminology, an instance) in the system. The frame (Z.120 terminology, parallel frame) represents the environment. Instances can interact with other instances or the environment by sending messages ($m0, m1, m2, m3$). The square labeled with a is an action performed by instance $i2$. The timing information is captured by the two rules below:

- For each message passing, the message output event precedes the corresponding message input event.
- For each vertical line representing an instance, the time progresses from top to bottom.

HMSCs can be constructed incrementally by referencing a MSC using its name. MSCs can be combined vertically, horizontally or alternatively. Various constructors for composing MSCs are: **alt**, **seq**, **par**, **opt**, **exc** and **loop**. Precise semantics are developed for these key words, e.g., **alt** is defined as *delayed choice*, denoted as $\mp$, and **par** are defined as *delayed parallel composition*, denoted as $\parallel$. Figure 2 is a simple example of HMSCs. Its semantics is captured by the process expression $(A \circ C)^* \circ (A \circ B)$, where $*$ means unbounded recursion and $\circ$ means sequential composition.

Various semantic models are developed for MSCs. Examples are operational semantics for MSCs based on process algebra [1,15], Petri nets [31], automata, etc. The informal MSC semantics and formal process algebra semantics for MSC [15] are adopted in this paper.

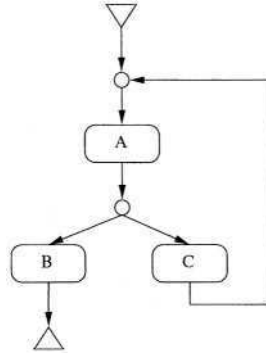


Fig. 2. A HMSC Example

3 Generate MSCs from TCOZ

MSCs is a simple graphical notation for capturing system components interaction. The process semantics MSCs is closely associated with the untimed trace aspects of the TCOZ semantics. Therefore, our first task is to build the untimed trace model for TCOZ that filters out unrelated issues i.e. timed refusals.

3.1 Trace Model for TCOZ

Semantic models for TCOZ is the infinite timed-states model [18] which extends the TCSP's infinite timed-failure model. MSCs, on the other hand, can be referred as 'untimed'. It deliberately abstracts from the precise times when events happen. Instead, MSC uses timers to capture basic timing information (timeout or timer reset). The untimed trace model in this section is mainly based on the trace models for CSP [14].

A TCOZ event may be either an update event, a simple synchronization, a channel communication, or a termination event.

$$\begin{aligned}
 U &== [v: \mathbb{I}; d: \text{VAL}] \\
 S &== [c: \Sigma] \\
 C &== [c: \Sigma; d: \text{VAL}] \\
 \Sigma &== \text{update}(\langle U \rangle) | \text{sync}(\langle S \rangle) | \text{com}(\langle C \rangle) | \checkmark
 \end{aligned}$$

To make use of the timer constructs in MSC, we extend TCOZ event with a special event *wait*(*d*), where *d* can be any real number. Basically, this event delays a process by time *d*. To filter out the unnecessary timing information, we simplify semantic functions for each TCOZ process expression. Given $[\Sigma]$ representing events, $[ZE]$ representing Z expressions, $[ZS]$ representing Z schemas, $[NAME]$ representing all valid character strings, a TCOZ process expression is defined as [19]:

Table 1. Simplified Trace Model of TCOZ based on [19]

Predicate	Remarks
$\mathcal{H}(\text{STOP}) = \{\langle \rangle\}$	(Tr-1)
$\mathcal{H}(\text{CHAOS}) = \{(\Sigma \setminus \{\checkmark\})^*\}$	(Tr-2)
$\mathcal{H}(\text{WAIT } ZE) = \{\langle \text{wait}(ze) \rangle\}$	(Tr-3)
$\mathcal{H}(ZS \bullet TZE) = \mathcal{H}(TZE) \cup \{\langle \rangle\}$	(Tr-4)
$\mathcal{H}(c.a \rightarrow TZE) = \{\langle c.a \rangle \hat{\sim} u \mid u \in \mathcal{H}(TZE)\} \cup \{\langle \rangle\}$	(Tr-5)
$\mathcal{H}(TZE_1 \mid TZE_2) = \mathcal{H}(TZE_1) \cup \mathcal{H}(TZE_2)$	(Tr-6)
$\mathcal{H}(TZE_1 \nabla TZE_2) = \{s \hat{\sim} t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-7)
$\mathcal{H}(TZE_1 \parallel TZE_2) = \mathcal{H}(TZE_1) \cap \mathcal{H}(TZE_2)$	(Tr-8)
$\mathcal{H}(TZE_1 TZE_2) = \bigcup \{s t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-9)
$\mathcal{H}(TZE_1 X TZE_2) = \bigcup \{s X t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-10)
$\mathcal{H}(TZE_1; TZE_2) = \{\mathcal{H}(TZE_1) \cap (\Sigma \setminus \{\checkmark\})^*\} \cup \{s \hat{\sim} t \mid s \in \mathcal{H}(TZE_1) \wedge t \in \mathcal{H}(TZE_2)\}$	(Tr-11)
$\mathcal{H}(\mu X \bullet TZE) = \bigcup_{n \geq 0} \mathcal{H}(F^n(\text{STOP}))$	(Tr-12)

$$\begin{aligned}
TZE ::= & \text{ref}\langle NAME \rangle \mid \text{STOP} \mid \text{CHAOS} \mid \text{SKIP} \mid \text{WAIT}\langle ZE \rangle \mid \text{op}\langle ZS \rangle \mid \\
& (- \bullet -)\langle ZS \times TZE \rangle \mid (- \rightarrow -)\langle \Sigma \times TZE \rangle \mid (- \rightarrow -)\langle \Sigma \times ZE \times TZE \rangle \mid \\
& (- \sqcap -)\langle TZE \times TZE \rangle \mid (- \sqcup -)\langle TZE \times TZE \rangle \mid (- \parallel -)\langle TZE \times TZE \rangle \mid \\
& (\mu \bullet -)\langle NAME \times TZE \rangle \mid (-[-/-])\langle TZE \times \Sigma \times \Sigma \rangle \mid (-\backslash -)\langle TZE \times \mathbb{P}\Sigma \rangle \mid \\
& (- \nabla -)\langle TZE_1 \times TZE_2 \rangle \mid (- ||| -)\langle TZE \times TZE \rangle \mid (-; -)\langle TZE \times TZE \rangle \mid \\
& (- \triangleright \{-\} -)\langle TZE \times ZE \times TZE \rangle \mid (-[[-] -)\langle TZE \times \mathbb{P}\Sigma \times TZE \rangle
\end{aligned}$$

A trace is defined as a sequence of events. Given a set of events (Σ) , Σ^* denotes all possible traces can be composed by events in Σ . Function $\upharpoonright$ filters out a set of events from a trace.

$$\text{Trace} \hat{=} \text{seq } \Sigma$$

$ \begin{aligned} & _ \bullet _ : \mathbb{P}\Sigma \rightarrow \mathbb{P}\text{Trace} \\ & \hline & \forall e : \mathbb{P}\Sigma; tr : \text{Trace} \bullet \\ & \quad tr \in e^* \Leftrightarrow \text{ran } tr \subseteq e \end{aligned} $	$ \begin{aligned} & _ \upharpoonright _ : \text{Trace} \times \mathbb{P}\Sigma \rightarrow \text{Trace} \\ & \hline & \forall t_i, t_e : \text{Trace}, e : \mathbb{P}\Sigma \bullet \\ & \quad t_i \upharpoonright e = t_e \Leftrightarrow \\ & \quad \text{ran } t_e \cap e = \emptyset \wedge \\ & \quad (\text{head}(t_i) = \text{head}(t_e) \wedge \\ & \quad \quad \text{tail}(t_i) \upharpoonright e = \text{tail}(t_e)) \vee \\ & \quad (\text{head}(t_i) \neq \text{head}(t_e) \wedge \\ & \quad \quad \text{tail}(t_i) \upharpoonright e = t_e) \end{aligned} $

We define a function $\mathcal{H} : TZE \rightarrow \mathbb{P}\text{Trace}$, which returns the set of all possible traces given a TCOZ process expression. Table 1 illustrates the detailed definition for the function $\mathcal{H}$, which defines how to compute the set of all possible traces for a TCOZ process expression inductively. The definition of the function $\mathcal{H}$ is based on the denotational semantics of CSP [14,24].

STOP means deadlock and performs no events (Tr-1), while CHAOS can perform any event (Tr-2). WAIT ZE delays a process by ze time unit (Tr-3). The state-guard, is used to *block* or *enable* execution of an operation on the basis

of an object's local state (the instance's state) (Tr-4). In general, state-guard can be complex. In our trace model, it is treated as non-deterministic choice, which may introduce unexpected traces. This is not a problem for our work. Tr-5 captures the case that a process expression is guarded by some channel communication event. The only way to proceed is to perform the communication. Tr-6 covers both internal choice ($P \sqcap Q$) and external choice ($P \sqbox Q$). For internal choice, the choice is made upon the internal state of the system. While for external choice, the choice is made by the environment. From the system interactive (MSC) viewpoint, this distinction is irrelevant. Tr-7 captures that Q interrupts P . Tr-8 expresses that the two processes synchronize on all events. Tr-9 expresses that two processes run completely independently. Tr-10 expresses a general case of Tr-8 and Tr-9, i.e., instead of synchronizing all (or none) of the events, only events in the set X are synchronized.

$\begin{array}{l} \hline - - : \text{Trace} \times \text{Trace} \rightarrow \mathbb{P} \text{Trace} \\ \hline \forall a, b, c : \text{Trace} \bullet \\ c \in a b \Leftrightarrow \\ \#c = \#a + \#b \wedge \\ c \upharpoonright \text{ran } a = a \wedge \\ c \upharpoonright \text{ran } b = b \end{array}$	$\begin{array}{l} \hline - [-] - : \text{Trace} \times \mathbb{P} \Sigma \times \text{Trace} \rightarrow \mathbb{P} \text{Trace} \\ \hline \forall a, b, c : \text{Trace}; e : \mathbb{P} \Sigma \bullet \\ c \in a [e] b \Leftrightarrow \\ c \upharpoonright e = a \upharpoonright e = b \upharpoonright e \wedge \\ c \upharpoonright \text{ran } a = a \wedge \\ c \upharpoonright \text{ran } b = b \end{array}$

Tr-11 expresses sequential composition of process expressions. TZE_2 can only take control after TZE_1 successfully terminates. This definition of sequential composition is known as *strong sequential composition* [1]. The trace model for recursion (Tr-12) is a fixed point definition. The trace function for SKIP, $\triangleright \{t\}$, $\swarrow \{t\}$ can be derived using the following laws.

$$\begin{aligned} \text{Skip} &\triangleq \checkmark \rightarrow \text{STOP} \\ TZE_1 \triangleright \{t\} TZE_2 &= TZE_1 \sqcap (\text{WAIT } t; TZE_2) \\ TZE_1 \swarrow \{t\} TZE_2 &= TZE_1 \nabla (\text{WAIT } t; TZE_2) \end{aligned}$$

Example: The set of traces for a process expression can be efficiently identified by applying function $\mathcal{H}$ recursively. We take the process *ButtonPushing* as an example.

$$\begin{aligned} \mathcal{H}(\text{ButtonPushing}) &= \mathcal{H}(\text{button?1} \rightarrow ([\text{dim} > 0] \bullet \text{TurningOff} \sqcap \\ &\quad [\text{dim} = 0] \bullet \text{TurningOn})) \\ &= \{ \langle \text{button?1} \rangle \frown u \mid u \in \mathcal{H}([\text{dim} > 0] \bullet \text{TurningOff} \sqcap \\ &\quad [\text{dim} = 0] \bullet \text{TurningOn}) \} \cup \{ \langle \rangle \} \\ &= \{ \langle \text{button?1} \rangle \frown u \mid u \in \mathcal{H}([\text{dim} > 0] \bullet \text{TurningOff}) \cup \\ &\quad \mathcal{H}([\text{dim} = 0] \bullet \text{TurningOn}) \} \cup \{ \langle \rangle \} \\ &= \{ \langle \text{button?1}, \text{TurningOff} \rangle, \langle \text{button?1}, \text{TurningOn} \rangle, \langle \text{button?1} \rangle, \langle \rangle \} \end{aligned}$$

3.2 Link Traces with BMSCs

Given one active object, we can identify the set of possible traces by applying the $\mathcal{H}$ function to the MAIN process. A trace can be transformed to a BMSC by identifying update events in TCOZ with MSC atomic local actions and identifying channel communications in TCOZ with message passing in MSCs.

In the previous subsection, a TCOZ event is defined as either an update event, a simple synchronization, a channel communication, or a termination event, or a $\text{WAIT}(d)$ event.

- Update events are distinguished from the others in the way that they do not require the cooperation of the environment or other processes. They perform on a single instance. A MSC local action is defined as an orderable single instance event requiring no cooperation from environment. Update events are identified with local actions in MSCs.
- Synchronization and channel communication do require cooperation either from environment or other processes. Channel communications in TCOZ are identified with message passings in MSCs.
MSCs support both synchronous and asynchronous message passing, channel communication in TCOZ is identified with synchronous message passing (message passing with a 0-capacity buffer).
- The special *wait* event in TCOZ is identified with the timer event in MSCs. In particular, it is identified with a timer set event in MSCs and consequently associated with a timeout or reset event.

Example: Figure 3 (the generated BMSC from TCOZ by applying $\mathcal{H}$ function) represents a possible scenario of the process *ControlledLight*. Initially the light is off. Starting with MAIN, the process *DimChange* is executed. A message input event *dimmer?n* takes place. At that moment *on* is false, no action is taken. Process *ButtonPushing* is then activated by a message input event from channel *button*. Action *TurningOn* is invoked. After that, no event occurs.

3.3 Project TCOZ Specifications to HMSCs

Due to unbounded recursion (iteration) and non-determinism, possible traces (and generated BMSCs) for some systems could be numerous or even infinite¹. HMSCs offer various constructive operators to compose MSCs in a hierarchical, iterating and nondeterministic way. In this section, we link TCOZ specification with HMSCs by identifying various operators in TCOZ with constructs in HMSCs.

The body of a TCOZ class is essentially a system of simultaneous equations defining a collection of operations (processes). Each equation consists of a name [*NAME*] and a TCOZ process expression.

¹ For LCS, 600+ traces are generated if we unfold recursions 5 times.

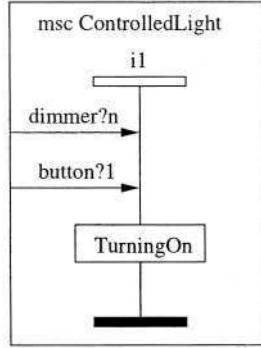


Fig. 3. BMSC: ControlledLight

A TCOZ class is identified with a MSC document². A TCOZ process expression is identified with a MSC. If a TCOZ process invokes other process expressions (by name), the process expression name will be identified with a MSC reference. A MSC reference expression is defined as the following,

$$\begin{aligned}
 MRE ::= & \text{ref} \langle \langle NAME \rangle \rangle \mid \epsilon \mid \delta \mid (- \circ -) \langle \langle MRE \times MRE \rangle \rangle \mid \\
 & (- \mp -) \langle \langle MRE \times MRE \rangle \rangle \mid (- \parallel -) \langle \langle MRE \times MRE \rangle \rangle \mid \\
 & (-)^{\otimes} \langle \langle MRE \rangle \rangle \mid (-)^{\infty} \langle \langle MRE \rangle \rangle
 \end{aligned}$$

Given a MSC reference expression, Function $\mathcal{G} : MRE \rightarrow \mathbb{P}Trace$ returns a set of traces capturing all possible behaviors of the process. In [15], semantics of various constructs of MSCs are specified by sets of deduction rules. A deduction rule is of the form $\frac{H}{C}$ where H is a set of premises and C is the conclusion. Each individual premise and conclusion are of the form $s \xrightarrow{a} s'$ or $s \downarrow$ for arbitrary $s, s' \in MRE$ and $a \in A$, where A denotes all events represented by atomic actions in MSCs, for example, message input, message output, local action and timer events. Following those deduction rules, the trace models $\mathcal{G}$ can be constructed.

$$\left| \begin{array}{l} \mathcal{P} : TZE \rightarrow MRE \\ \hline \forall s : TZE; t : MRE \bullet \mathcal{P}(s) = t \Rightarrow \mathcal{H}(s) = \mathcal{G}(t) \end{array} \right|$$

A projection function $\mathcal{P}$ from TCOZ process expression to MSC reference expression can be established when the set of possible traces for the TCOZ process expressions is identical with those for the MSC reference expressions.

STOP and SKIP. In TCOZ, STOP means deadlock and no communications. SKIP performs no action except for successful termination. The two basic constants, denoted as δ and ϵ , also play the same role in the process semantics for

² A MSC document consists a set of MSCs.

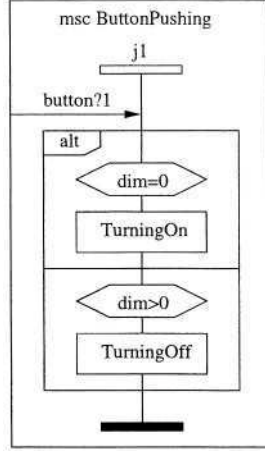


Fig. 4. Transformation: Choice

MSCs. No deduction rule is associated with δ . The rule associated with ε is $\varepsilon \downarrow$, meaning successful termination.

$$\begin{aligned}\mathcal{G}(\delta) &= \{\langle \rangle\} \\ \mathcal{G}(\varepsilon) &= \{\langle \checkmark \rangle\} \cup \{\langle \rangle\}\end{aligned}$$

Graphically, SKIP is mapped to a MSC containing no event.

Choice. In [15], a structural operation *delayed choice* is denoted by $\mp$. Its semantics is expressed in the following rules:

$$\begin{array}{ccc} \frac{x \downarrow}{x \mp y \downarrow} [DC1] & \frac{y \downarrow}{x \mp y \downarrow} [DC2] & \frac{x \xrightarrow{a} x', y \not\xrightarrow{a}}{x \mp y \xrightarrow{a} x'} [DC3] \\ \\ \frac{x \not\xrightarrow{a}, y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} y'} [DC4] & \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \mp y \xrightarrow{a} x' \mp y'} [DC5] \end{array}$$

The rules *DC1* and *DC2* express that the delayed choice of the two processes has the option to terminate if and only if at least one of the alternatives has this option. *DC3* and *DC4* express that the delayed choice will behave as one of the options given that some initial event of this option takes place. *DC5* captures the idea that in case both of the alternatives are enabled, the choice is delayed.

We can verify that the set of possible traces of $x \mp y$ is the union of traces of x and y , i.e., $\mathcal{G}(x \mp y) = \mathcal{G}(x) \cup \mathcal{G}(y)$. Assume $s \in \mathcal{G}(x \mp y)$, if $\text{head}(s)$ is an event that can be performed by process x and not by process y , *DC3* applies, the set of all such traces is $\{\langle \text{head}(s) \rangle \wedge u \mid u \in \mathcal{G}(x')\}$, which is a subset of $\mathcal{G}(x)$. If $\text{head}(s)$ is an event that can be performed by y and not by x , rule *DC4* applies, following

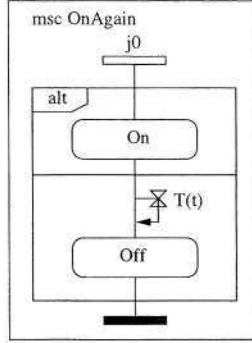


Fig. 5. Transformation: Timeout

the same argument, we can verify that the set of such traces is a subset of $\mathcal{G}(y)$. If $head(s)$ can be performed by both x and y , $DC5$ applies, the set of possible traces is $\{\langle head(s) \rangle \sim u \mid u \in \mathcal{G}(x' \mp y')\}$, which is a subset of $\mathcal{G}(x) \cup \mathcal{G}(y)$. Since s is an arbitrary trace in the set $\mathcal{G}(x \mp y)$, we conclude that $\mathcal{G}(x \mp y)$ is a subset of $\mathcal{G}(x) \cup \mathcal{G}(y)$. By the similar construction, we can conclude that for any trace in $\mathcal{G}(x) \cup \mathcal{G}(y)$, it is also in $\mathcal{G}(x \mp y)$. This completes the construction of the choice operator. In term, other mappings can be formulated in a similar way.

Figure 4 illustrates how the transformation is done graphically for the process *ButtonPushing* in *ControlledLight* class. Keyword **alt** (short for alternative) is used to denote *delayed choice* graphically.

Timeout. In TCOZ, $P \triangleright \{t\}Q = P \sqcap (\text{Wait } t; Q)$. By identifying external choice with MSC delayed choice and $\text{WAIT } t$ with timer events, timeout can be identified with MSCs constructed by a delayed choice between P and Q with a timeout event as the initial event of Q . In the room controller model of the LCS example, given process $OnAgain \hat{=} (motion?1 \rightarrow On) \triangleright \{t\} Off$, it is transformed to the MSC in Figure 5.

Interleaving. Delayed parallel composition, denoted by $\parallel$ in MSC, is rewritten as $\parallel_m$ to avoid confusion. Delayed parallel composition defines the interleaving operator, i.e., no synchronization is required and processes can interleave freely³. Interleaving in TCOZ ($\parallel$) is identified with delayed parallel composition in MSCs.

$$\mathcal{G}(P \parallel_m Q) = \bigcup \{s \parallel t \mid s \in \mathcal{G}(P) \wedge t \in \mathcal{G}(Q)\}$$

Synchronization. In TCOZ, all dynamic interactions between active objects must take place through the CSP channel communication mechanism. All synchronization is done by message passing through channels.

³ Refer to [15] for detailed definition of delayed parallel composition.

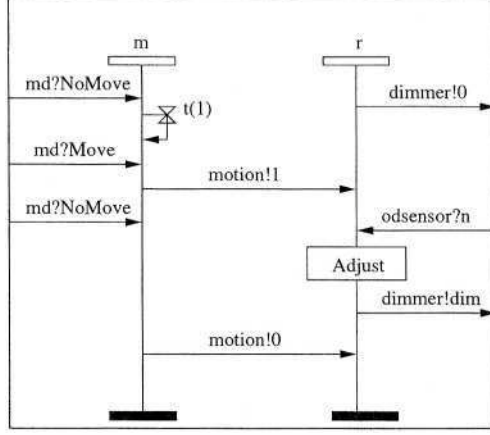


Fig. 6. Transformation: Synchronization

No synchronization construct is defined in MSCs. Graphically, given two synchronizing MSCs (P and Q), the composed MSC is constructed by putting the MSCs in the same parallel frame and connecting corresponding message output and message input events. By adopting the view that message passing are synchronized, the newly constructed MSC represents the set of traces as

$$\bigcup \{s \parallel [X]t \mid s \in \mathcal{G}(P) \wedge t \in \mathcal{G}(Q)\}$$

where X denotes all events on the sharing channel.

In the *LCS* class, active object m (motion detector) shares the channel *motion* with the active object r (room controller). A possible trace for m would be

$$\langle md?NoMove, wait\ 1, md?Move, motion!1, md?NoMove, motion!0, \dots \rangle$$

A matching trace for r must contain the same events on channel *motion*. For example, a matching sequence would be

$$\langle dimmer!0, motion?1, odsensor?n, olight := n, dimmer!dim, motion?0, \dots \rangle$$

The interaction can be visualized as Figure 6.

By constructing the composed MSCs as above, we can make use of the full power of MSC's partial ordering property. That is, to leave the order of single instance events from different instances unspecified. Thus, one MSC is capable of representing a set of scenarios.

Sequential Composition. Sequential composition in TCOZ is best described as *strong sequential composition*. *Strong sequential composition* of two processes x and y behaves like process x and upon termination of x it starts behaving like process y . No action from process y can be executed before x has the option

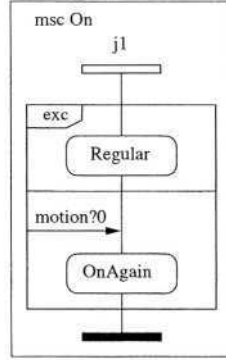


Fig. 7. Transformation: Interrupt

to terminate. In [15], a different approach, named *weak sequential composition*, denoted as $\circ$ is adopted to compose two MSCs vertically. The *weak sequential composition* allows the execution of actions from y before x has the option to terminate.

All synchronizations in TCOZ are taken through channels, the ordering information of local actions from different instances are irrelevant. On the other hand, in case two MSCs only involve events on the same process, The *weak sequential composition* and the *strong sequential composition* are the same. Sequential composition in TCOZ is identified with sequential composition in MSCs.

Graphically, sequential composition of MSCs on the same instances is captured by putting the MSCs one below the other.

Interrupt. MSCs have a key word **exc** for representing exceptions, however there is no formal rules defined in [15]. We define the rules for **exc** (the symbol ∇_m is used instead) as follows.

$$\begin{array}{c}
 \frac{x \downarrow}{x \nabla_m y \downarrow} \qquad \frac{x \xrightarrow{a} x', y \xrightarrow{a} y'}{x \nabla_m y \xrightarrow{a} x' \nabla_m y} \qquad \frac{y \xrightarrow{a} y'}{x \nabla_m y \xrightarrow{a} y'}
 \end{array}$$

In the process $X \nabla Y$ (Y has an initial event e), any time e takes place, X is interrupted and control transfers to Y . Interrupt in TCOZ is identified with ∇_m in MSCs with e as the initial event of the interrupting process. For example, in *RoomController*,

$$On \hat{=} Regular \nabla motion?0 \rightarrow OnAgain$$

can be transformed to MSC as in Figure 7.

Besides the projection links above, the rest constructs in TCOZ can be transformed to constructs in MSCs in obvious ways. TCOZ recursion can be resolved as iteration and interpreted by a sequence of sequential composition, which can

be generalized as the iteration operator $\oplus$. TCOZ *state-guard* is identified with *condition* in MSCs.

3.4 Generate Test Requirements

Test requirements can be used to develop test cases, test oracles and test drivers in a system development. Specification based testing can play an important role in software engineering [23,27]. TCOZ specification based testing can be based on the generated MSCs (execution scenarios). Our goal is to support automatic generation of test requirements.

Four steps are essential, which are all based on the MSCs generated.

- Starting with a HMSC, one can expand the HMSC into a set of BMSCs. In the recursion case, at least one iteration should be covered by the expanded BMSCs.
- Upon creation of an instance, TCOZ class initial state condition is instrumented as an assertion at the start of the BMSC.
- For each instance in the system, TCOZ class invariants are instrumented as assertions before and after each action on the BMSC instance.
- The pre/post-conditions of TCOZ operations is transformed to assertions at the entry/exit of the corresponding MSC actions.

We will illustrate these steps by taking the *ControlledLight* class as an example. First, we resolve the unbound recursion in MAIN process by performing it once. From Figure 4, we identify two event sequences from both *DimChange* and *ButtonPushing* because of the delayed choice operator. The testing requirements for *ControlledLight* is captured by Figure 8 (an expanded BMSC with Assertions). Assertions are placed in the dash line square-boxes. The event sequence for *DimChange* containing SKIP is dropped since SKIP represents an empty sequence of events.

4 Automation

The translation process can be automated by employing XML/XSL technology. In our previous work [28], a XML interchange format ZML for Z family languages, i.e., Z/Object-Z/TCOZ, has been defined using XML Schema. MSC also offer a standard text representation for the graphical notations. In this work, an automatic transformation tool is developed in Java to project TCOZ models (in ZML) into MSCs (in standard text format).

Building on the strength of ZML, our tool makes use of XML parser Xerces to extract information from TCOZ specifications. For example, the following is a part of the *ControlledLight* class model in ZML.

```
<classDef>
  <name>ControlledLight</name>
  <inheritedClass><name>Light</name></inheritedClass>
  <state>...</state>
  <operation>
```

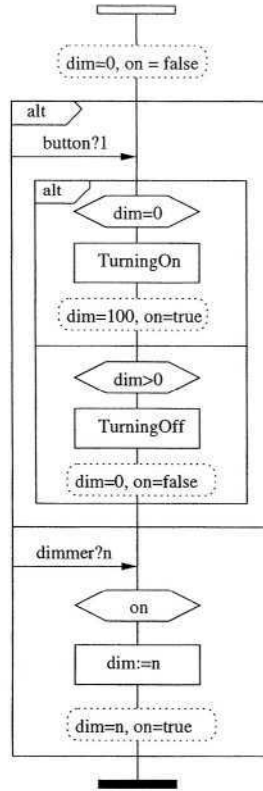


Fig. 8. Test Requirements

```

<name>Main</name>
<processExpr>
  <mu>N</mu>
  <processExpr>
    <processExpr>
      <simpleProExp>ButtonPushing</simpleProExp>
    </processExpr>
    <proConnSym>externalChoice</proConnSym>
    <processExpr>
      <simpleProExp>DimChange</simpleProExp>
    </processExpr>
  </processExpr>
</processExpr>
</operation>
</classDef>

```

The automatic transformation is achieved by first implementing a ZML parser, which will take in a specification model in ZML and build a virtual model in the memory. This ZML parser can be reused for other projection tools (e.g. the transformation from TCOZ to Timed Automata for timing analysis).

A trace generation module is built to automatically generate all possible traces for a specification model, each trace can be transformed to a BMSC by syntax rewriting. In the case of unbounded recursion, users may provide the expected number of iterations.

An MSC interface is built according to the MSC document structure, e.g., each MSC document contains multiple MSCs and each MSC contains one or more instances and etc. A transformation module is built to get information from the ZML parser, apply the right transformation rules (specified in Section 3) and feed the outcome of the transformation to the MSC interface. The transformation rules are used as a design document and guide the construction of various transformation algorithms in the implementation.

The outcome of our transformation tool is Z.120 standard text representation of MSCs, which is ready to be taken as inputs for various tool support for MSCs. For example, the above XML representation of MAIN operation in *ControlledLight* is transformed to a HMSC as:

```
mhc ControlledLight;
instance i1;
loop begin;
alt begin;
reference ButtonPushing;
alt;
reference DimChange;
alt end;
loop end;
endinstance;
endmhc;
```

Reuse for Timed Viewpoint Projection

The same strategy can be applied for implementing various transformation tools. For example, for the timing analysis purpose, a TCOZ specification can be transformed into Timed Automata (we are currently building this tool), the same ZML parser can be reused and we only need to build a Timed Automata interface and a new transformation module.

5 Conclusion

In this paper, we investigate the semantic based transformation from TCOZ to MSCs and present a tool to automatically generate MSCs from the TCOZ specifications.

An untimed trace model for TCOZ is introduced to focus on the interaction viewpoints. Each possible trace is identified with a BMSC by linking TCOZ update events as MSC local actions and channel communication as synchronous message-passing. The projection from TCOZ to HMSCs is constructed based on linking the trace semantic models of TCOZ constructs with the process semantics of HMSC constructs.

By inserting appropriate TCOZ specification constraints (as assertions) into the generated MSCs, we further explore ways of generating system test requirements from TCOZ.

Acknowledgments. This work is supported by the A*STAR research grant *Formal Design Techniques for Reactive Embedded Systems*

References

1. J. C. M. Baeten and W. P. Weijland. Process Algebra. *Cambridge Tracts in Theoretical Computer Science*, 18(1), 1990.
2. C. Bolton and J. Davies. Activity Graphs and Processes. In W. Grieskamp, T. Santen, and W. Stoddart, editors, *Proceedings of IFM 2000*, pages 77–96. Springer, 2000.
3. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A Formal Framework for Viewpoint Consistency. *Formal Methods in System Design*, 21:111–166, September 2002.
4. P. J. Brooke and R. F. Paige. The Design of a Tool-Supported Graphical Notation for Timed CSP. In M. J. Butler, L. Petre, and K. Sere, editors, *Proc. Integrated Formal Methods 2002 (IFM'02)*.
5. M. Butler. csp2B: A Practical Approach To Combining CSP and B. In J. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, Lect. Notes in Comput. Sci., Toulouse, France, September 1999. Springer-Verlag.
6. C-A. Chen, S. Kalvala, and J. Sinclair. Generating b specifications from message sequence charts. In *St.Eve Workshop*, September 2003.
7. A. Coombes and J. A. McDermid. Using Diagrams to Give a Formal Specification of Timing Constraints in Z. In *Z User Workshop*, pages 119–130, 1992.
8. J. Davies and C. Crichton. Using State Diagrams to Describe Concurrent Behaviour. In *ICFEM 2003, LNCS 2885*, pages 105–125, 2003.
9. J. Davies and S. Schneider. A Brief History of Timed CSP. *Theoretical Computer Science*, 138, 1995.
10. Marcio S. Dias and Debra J. Richardson. Identifying Cause and Effect Relations between Events in Concurrent Event-Based Components. In J. Richardson, W. Emmerich, and D. Wile, editors, *The 17th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002.
11. R. Duke and G. Rose. *Formal Object Oriented Specification Using Object-Z*. Cornerstones of Computing Series. Macmillan, March 2000.
12. R. L. Feldmann, J. Munch, S. Queins, S. Vorwieger, and G. Zimmermann. Baselineing a Doman-Specific Software Development Process. Tech Report SFB501 TR-02/99, University of Kaiserslautern, 1999.
13. C. Fischer and H. Wehrheim. Model-Checking CSP-OZ Specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
14. C.A.R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.
15. ITU. *Message Sequence Chart(MSC)*, Nov 1999. Series Z: Languages and general software aspects for telecommunication systems.
16. Shaoying Liu, A. Jeff Offutt, Chris Ho-Stuart, Yong Sun, and Mitsuru Ohba. Sofl: A formal engineering methodology for industrial applications. pages 24–45, 1998.
17. B. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177, February 2000.
18. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(2):142–160, 2002.

19. B. Mahony and J. S. Dong. Deep Semantic Links of TCSP and Object-Z: TCOZ Approach. *Formal Aspects of Computing*, 13(1):142–160, 2002.
20. M. Y. Ng and M. Butler. Tool Support for Visualizing CSP in UML. In C. George and H. Miao, editors, *International Conference on Formal Engineering Methods (ICFEM'02)*, pages 287–298. LNCS, Springer-Verlag, October 2002.
21. B. Nuseibeh, J. Kramer, and A. Finkelstein. A Framework for Expressing the Relationships Between Multiple Views in Requirement Specifications. *IEEE Trans. Software Eng.*, 20(10):760–773, October 1994.
22. L. Petre and K. Sere. Developing Control Systems Components. In W. Grieskamp, T. Santen, and B. Stoddart, editors, *IFM'00: Integrated Formal Methods*, Lect. Notes in Comput. Sci. Springer-Verlag, October 2000.
23. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-Based Test Oracles for Reactive Systems. In *International Conference on Software Engineering*, pages 105–118, 1992.
24. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
25. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
26. G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems - an Integration of Object-Z and CSP. *Formal Methods in System Design*, 18:249–284, 2001.
27. P. Stocks and D. Carrington. A Framework for Specification-based Testing. *IEEE Trans. Software Eng.*, 22(11):777–793, 1996.
28. J. Sun, J. S. Dong, J. Liu, and H. Wang. A Formal Object Approach to the Design of ZML. *Annals of Software Engineering*, 13:329–356, 2002.
29. K. Taguchi and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M. Hinchey and S. Liu, editors, *the IEEE International Conference on Formal Engineering Methods (ICFEM'97)*, pages 283–292, Hiroshima, Japan, November 1997. IEEE Press.
30. H. Treharne and S. Schneider. Using a Process Algebra to Control B Operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *IFM'99: Integrated Formal Methods, York, UK*. Springer-Verlag, June 1999.
31. K.M. van Hee. *Information Systems Engineering: A Formal Approach*. Cambridge University Press, Cambridge, 1994.
32. J. Woodcock and A. Cavalcanti. The Steam Boiler in a Unified Theory of Z and CSP. In J. He, Y. Li, and G. Lowe, editors, *The 8th Asia-Pacific Software Engineering Conference (APSEC'01)*, pages 291–298. IEEE Press, 2001.

UML to B: Formal Verification of Object-Oriented Models

K. Lano, D. Clark, and K. Androutsopoulos

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK.
{kcl, david, kelly}@dcs.kcl.ac.uk Phone: 0207 848 2832, Fax: 0207 848 2851

Abstract. The integration of UML and formal methods such as B and SMV provides a bridge between graphical specification techniques usable by mainstream software engineers, and precise analysis and verification techniques, essential for the development of high integrity and critical systems. In this paper we define a translation from UML class diagrams into B, which is used to verify the consistency of UML models and to verify that expected properties of these models hold.

Keywords: UML, B, UML-RSDS, Graphical Specifications.

1 Introduction

In the RSDS method [10], a small subset of UML statechart diagrams was used as a specification and design notation for reactive systems. From such specifications, executable code could be synthesised, and B [1] and SMV descriptions produced which allowed the analysis of static and temporal properties of the system. This approach is adequate for small control systems without dynamic reconfiguration. However for more general control systems, and for other types of critical system such as e-commerce, the expressibility of a larger subset of UML is required.

This paper therefore extends RSDS by:

1. Translating general UML class diagram structures into B, including diagrams involving inheritance.
2. Translating a subset of OCL into B.
3. Synthesising the code of methods from OCL constraints.

The extended RSDS notation incorporating UML class diagrams is referred to as UML-RSDS in the following.

2 UML-RSDS Specifications

UML-RSDS specifications consist of:

1. A UML class diagram, including constraints attached to operations, classes and (sets of) associations;
2. Statemachines attached to attributes of classes in the class diagram.

Attributes are stereotyped as *sensor*, *internal*, *derived* or *actuator*: state changes on sensor attributes are used as input events to the system, these then lead to changes to internal and derived attributes of the same and/or different classes, and then to actuator attribute changes, which effect the output of the system. The decoration ? indicates a sensor attribute and ! an actuator, the conventional / decoration is used for derived attributes. These stereotypes are used to guide the translation of classes, as in the approach of [20]. For general software systems, sensors represent inputs to the system (eg, HTTP requests in a web-based system) and actuators outputs from the system (eg, generated web pages).

Table 1. Syntax of LOCA Expressions

$\langle \text{value} \rangle$	$::= \langle \text{ident} \rangle \mid \langle \text{number} \rangle \mid \langle \text{string} \rangle \mid \langle \text{boolean} \rangle$
$\langle \text{objectref} \rangle$	$::= \langle \text{ident} \rangle \mid \langle \text{objectref} \rangle . \langle \text{ident} \rangle \mid \langle \text{objectref} \rangle [\langle \text{expression} \rangle]$
$\langle \text{factor} \rangle$	$::= \langle \text{value} \rangle \mid \{ \langle \text{valueseq} \rangle \} \mid \langle \text{objectref} \rangle \mid \langle \text{factor} \rangle \text{op1} \langle \text{factor} \rangle$
$\langle \text{expression1} \rangle$	$::= \langle \text{factor} \rangle \text{op2} \langle \text{factor} \rangle$
$\langle \text{expression} \rangle$	$::= \langle \text{expression1} \rangle \mid \langle \text{expression1} \rangle \text{op3} \langle \text{expression1} \rangle$
$\langle \text{stateinvariant} \rangle$	$::= \langle \text{expression} \rangle \mid \langle \text{expression} \rangle \Rightarrow \langle \text{expression} \rangle$

The constraint language we use, LOCA (Logic of objects, constraints and associations) is a notational variant of OCL 2.0 [15]. Table 1 gives the formal syntax of single-state invariants in LOCA. A $\langle \text{valueseq} \rangle$ is a comma-separated list of $\langle \text{value} \rangle$ s. An op1 is one of +, −, *, /, \/, /\. An op2 is one of =, /=, <, >, <=, >=, :, <:, />:. An op3 is one of &, or. Identifiers are either class names, function names, class features (attribute or role names), elements of enumerated types, or represent variables or constants (if in upper case). Variables are implicitly universally quantified over the entire formula.

There are the following omissions from OCL and specific interpretations:

- When used in constraints attached to associations, the *context* of the LOCA formula is the set of all object pairs linked by these associations. This means that, provided attributes of the connected classes have distinct names, in many formulas we can omit any quantifiers or reference to specific objects completely. For example the constraint C1:

$$\text{locn} = \text{occupied} \Rightarrow \text{rte} = \text{occupied}$$

in Figure 2 represents the inter-class invariant

$$\forall l : \text{Location}; r : \text{Route} \cdot (r, l) \in \text{Route\_Location} \wedge l.\text{locn} = \text{occupied} \Rightarrow r.\text{rte} = \text{occupied}$$

It could also be expressed as an invariant of *Route*:

$$\text{occupied} : \text{path}.\text{locn} \Rightarrow \text{rte} = \text{occupied}$$

This version is more biased towards a particular implementation where *Route* is responsible for maintaining the constraint, so the first version is preferred in UML-RSDS.

- Metamodel features of OCL are omitted: *oclIsTypeOf*, *oclIsKindOf*, *oclIsNew*, *oclAsType*, *allInstances*, *OclType*, *oclInState*, *OclState*, *OclAny*, *OclExpression*.
x.oclIsKindOf(t) is expressible as $x : t$ for class names t . *x.oclInState(s)* is expressible by $x.att = s$ where s is a state of the statemachine attached to attribute *att*. *C.allInstances()* is expressed by the name C by itself.
- The procedural operator *iterate* is omitted.

OCL navigation expressions, including implicit flattening of collections, can be used in LOCA.

Some concepts from B have been adopted, for example, the use of set equality to avoid the use of quantifiers:

$$r.path = l$$

for a specific route r and location l in Figure 2 has semantics $path(r) = \{l\}$.

The result is a pure logical language which allows more concise expression of inter-class constraints than conventional OCL, where constraints are normally only attached to operation pre and post conditions and to individual classes. Such inter-class constraints are essential in abstract implicit specifications of system behaviour.

2.1 UML-RSDS Specification of Railway Signalling System

An example of a UML-RSDS specification for a railway signalling system [5, 11] is shown in Figure 2. This system supports dispatchers in organising the set-up and cancellation of routes and the blocking of parts of the network (eg, for maintenance) whilst enforcing safety rules such that at most one train can occupy a given route.

The railway network consists of a set of *locations*, each of which has an occupancy detector, and may also have a signal and/or a switch (Figure 1). Each switch may be in either the normal (straight through) position or reverse (directing the train to the side) position, or neither. A train can only travel over the point if it is in the normal or reverse position. Each switch has sensors *swn*, *swr* to detect its position, and motor *swset* to change its position. Each signal may be set to clear (allowing full speed) or half, or stop. The network is divided into *routes*, which consist of a sequence of adjacent track locations extending from a location containing a signal, the *lead signal* of the route, to a location immediately before the next signal in the same direction. In Figure 1 the four routes have been indicated.

The dispatchers can issue commands of the form *cmd locn*, where *locn* is the number of the location the command is intended for. *cmd* may be one of the following:

- *sigclr* – set the signal clear;

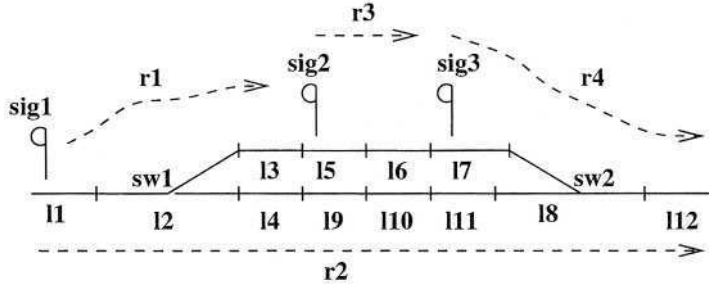


Fig. 1. Example Track Layout

- *sigstp* – set the signal to stop;
- *sigblk* – set the signal blocked;
- *sigubk* – unblock the signal;
- *swrev* – set the switch to reverse;
- *swnor* – set the switch to normal;
- *swblk* – block switch;
- *swubk* – unblock switch;
- *tkblk* – block track;
- *tkubk* – unblock track.

There may be checks required before a command can be accepted, for example a *sigclr* command can only be obeyed if the route associated with the signal is unoccupied, not blocked, none of its locations are used in another active route, none of the switches in the route are blocked, and they are all in a reverse or normal state, and the signal is not blocked. Alarms are raised if an invalid command is issued.

Figure 2 shows the abstract analysis model of the system as a UML-RSDS class diagram.

Some example constraints are:

1. C1 “If any location in a route is occupied, the route is occupied”

$$locn = occupied \Rightarrow rte = occupied$$

2. C2 “If any location in a route is blocked, so is the route”:

$$locstatus = blocked \Rightarrow routestatus = blocked$$

Similarly if any signal in the route is blocked, or if any switch in the route is blocked.

3. C3 “If a route is not ready (some switch is in neither the normal or reverse position) then it is not traversable”:

$$ready = false \Rightarrow traversable = false$$

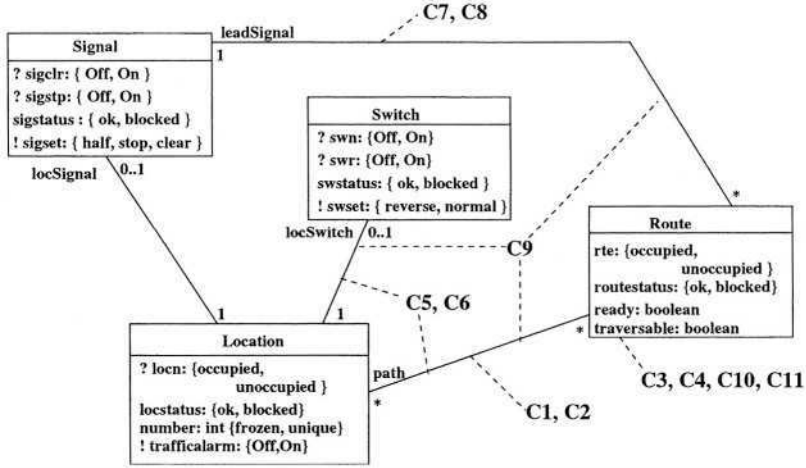


Fig. 2. Abstract Analysis Class Diagram

4. C5 “A route cannot be entered if some switch is not set to either normal or reverse”:

$$\begin{aligned} swr = Off \ \& \ swn = Off \Rightarrow ready = false \\ ready = false \Rightarrow leadSignal.sigset = stop \end{aligned}$$

5. C7 “If a route is occupied, its signal must be set to stop”:

$$rte = occupied \Rightarrow sigset = stop$$

6. C8 “If a route is blocked, its signal must be set to stop”:

$$routestatus = blocked \Rightarrow sigset = stop$$

Some operation specifications describing the effect of operator commands are as follows, where $AX(P)$ denotes that P holds in the next state, ie, at completion of the reaction to the event. This corresponds to P being a postcondition constraint of the operation in standard OCL notation.

– *sigstp*:

$$sigstp \ \& \ sigstatus = ok \Rightarrow AX(sigset = stop)$$

– *swnor*:

$$\begin{aligned} swnor \ \& \ rte = unoccupied \ \& \ swstatus = ok \ \& \\ routestatus = ok \Rightarrow AX(swset = normal) \end{aligned}$$

- The blocking and unblocking commands have no validation checks, and simply modify the values of the *locstatus*, *swstatus* and *sigstatus* variables as appropriate (and therefore also possibly the values of *routestatus* and the state of signals).

Finally we can express environmental constraints whose failure will trigger the creation of a traffic alarm for the location concerned, using the RSDS fault detection architecture [10].

- A switch is failed if its *swn* and *swr* sensors are both set:

$$swn = On \Rightarrow swr = Off$$

- Similarly for signals:

$$sigclr = On \Rightarrow sigstp = Off$$

- A location should not become occupied if it is blocked:

$$locnoccupied \Rightarrow locstatus = ok$$

- A route should only be entered by its initial location:

$$locnoccupied \ \& \ rte = unoccupied \Rightarrow locSignal = leadSignal$$

The usual rules of $\Rightarrow$ -transitivity and contraposition apply to LOCA constraints: if $A \Rightarrow B$ is a system constraint on the set r of associations, and $B \Rightarrow C$ is on the set s of associations, so are $A \Rightarrow C$ on set $r \cup s$, and $not(B) \Rightarrow not(A)$ on set r , provided that the names of all attributes in the classes are unique in the set of connected classes.

3 Translation from UML-RSDS to B

We use the B language [9] as a formal notation for expressing the precise semantics of a UML-RSDS model and for reasoning about this model. The translation makes explicit the meaning of the LOCA constraints of the model, and respects the structure of the model, with individual classes being represented by distinct machines in the translation.

3.1 Translation of Types

An enumerated type $T = \{v_1, \dots, v_n\}$ is translated to a corresponding *SET* in B. The enumerated types of attributes in the class diagram (corresponding to the states of their statemachines) are defined in the *SystemTypes* B machine, this is then accessed via *SEES* by all other generated B components.

The OCL *Boolean* type is interpreted by *BOOL* (defined as the set $\{TRUE, FALSE\}$ in the *Bool_TYPE* machine) in B. *String* is interpreted by *STRING* in B. The OCL *Integer* type is translated to B's *INT*. This translation is inexact as the OCL type is the mathematical set of integers whilst the B type is bounded.

The OCL *Real* type is not translatable to B.

A class type C is translated to a type C_OBJ in *SystemTypes* representing all possible instances of C , and a variable $cs \subseteq C\_OBJ$ representing the set of existing instances of C . This variable will be located in the B machine that represents C .

3.2 Translation of Classes and Associations

For a class C with attributes $att_1 : T_1$ to $att_n : T_n$ and roles $role_1$ of cardinality c_1 to class D_1 , ..., $role_m$ of cardinality c_m to class D_m (Figure 3), we define a machine $\mathcal{B}(C)$ using the usual translation [9] of attributes and associations to maps:

```

MACHINE C
SEES SystemTypes
USES D1, ..., Dm
VARIABLES cs, att1, ..., attn, role1, ..., rolem
INVARIANT cs <: C_OBJ &
    att1: cs --> T1 & ... & attn: cs --> Tn &
    role1: cs --> S1 & ... & rolem: cs --> Sm
INITIALISATION
    cs := {} || att1 := {} || ... || attn := {} ||
    role1 := {} || ... || rolem := {}
OPERATIONS
    cx <-- new_C(att1x,...,attnx,role1x,...,rolekx) =
        PRE att1x: T1 & ... & attnx: Tn &
            role1x: P1 & ... & rolekx: Pk & cs /= C_OBJ
        THEN
            ANY oox WHERE oox: C_OBJ - cs
            THEN
                cs := cs \ { oox } ||
                att1(oox) := att1x ||
                ...
                attn(oox) := attnx ||
                role1(oox) := role1x || /* for cardinality 1 roles */
                ...
                rolek(oox) := rolekx ||
                role(oox) := {} || /* for roles with cardinality /= 1 */
                cx := oox
            END
        END;

    att1x <-- getatt1(cx) =
        PRE cx: cs THEN att1x := att1(cx) END;

    ...

    attnx <-- getattn(cx) =
        PRE cx: cs THEN attnx := attn(cx) END;

    role1x <-- getrole1(cx) =
        PRE cx: cs THEN role1x := role1(cx) END;

    ...

    rolemx <-- getrolem(cx) =
        PRE cx: cs THEN rolemx := rolem(cx) END;

```

```

setatt1(cx, val1) =
  PRE cx: cs & val1: T1 THEN att1(cx) := val1 END;

...

setattn(cx, valn) =
  PRE cx: cs & valn: Tn THEN attn(cx) := valn END;

setrole1(cx, role1x) =
  PRE cx: cs & role1x: S1 THEN role1(cx) := role1x END;

...

setrolem(cx, rolemx) =
  PRE cx: cs & rolemx: Sm THEN rolem(cx) := rolemx END
END

```

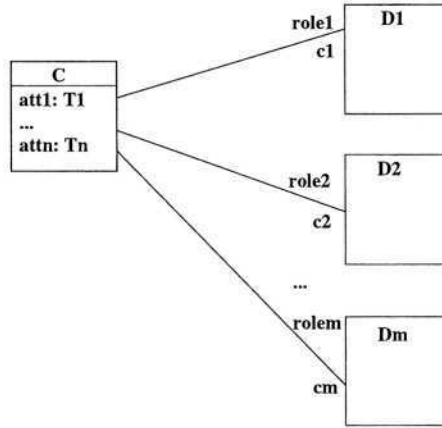


Fig. 3. Class with Attributes and Associations

The machine $\mathcal{B}(C)$ may need to SEE *Bool_TYPE*, *Int_TYPE* or *String_TYPE* if the corresponding data types are used as one of the T_i . If an initial value is specified in the class diagram for an attribute, or there is a statemachine with an initial state for the attribute, then this value is used instead of a parameter in *new_C*. If $role_j$ has ONE cardinality: $c_j = 1..1$, then S_j is simply the set d_j s of existing instances of D_j , and there is a parameter $role_jx$ and assignment $role_j(oox) := role_jx$ in the constructor operation. Otherwise, S_j is $\mathbb{F}(d_js)$ and the constructor assignment is $role_j(oox) := \{\}$, there is no parameter¹. The $role_1, \dots, role_k$ above are all the cardinality ONE roles, $role_i$, with P_i being the corresponding d_is .

¹ Provided c_j has a lower bound of zero.

For roles with a MANY (ie, $a..b$ with $a < b$, $a..*$ or $*$) cardinality there are also *addrole*(*cx*, *obj*), *removerole*(*cx*, *obj*) operations which add/remove *obj* from *role*(*cx*), and *unionrole*(*cx*, *objs*) and *subtractrole*(*cx*, *objs*) which add and remove sets *objs* of objects from *role*. Apart from $*$ cardinality roles, these have preconditions to ensure cardinality constraints.

Operations *setAllfeat*(*oos*, *val*), *addAllrole*(*oos*, *obj*), *removeAllrole*(*oos*, *obj*), *unionAllrole*(*oos*, *objs*), *subtractAllrole*(*oos*, *objs*) which perform multiple updates are also included, so they can be used by code synthesised from constraints.

For each association, invariants are derived from its role cardinalities. For example if *role* has target cardinality 1.. 2 then the invariant

$!cx.(cx: cs \Rightarrow \text{card}(\text{role2}(cx)): \{1,2\})$

is included in the machine for *C*. $!$ denotes the universal quantifier in B notation.

Table 2 shows the particular function types and extra invariants for the most common combinations of role cardinalities.

Table 2. Representation of Associations in B

Association	B role type	B invariants
$A_*-_*^rB$	$r : as \rightarrow \mathbb{F}(bs)$	
$A_{0..1}-_*^rB$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall a.(a \in as \Rightarrow r(a) \cap \text{union}(r[as - \{a\}]) = \emptyset)$
$A_1-_*^rB$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall a.(a \in as \Rightarrow r(a) \cap \text{union}(r[as - \{a\}]) = \emptyset)$ $\text{union}(r[as]) = bs$
A_*-1^rB	$r : as \rightarrow bs$	
$A_{0..1}-1^rB$	$r : as \rightarrow bs$	

If a UML-RSDS specification is purely declarative, that is, all functionality is specified implicitly via class and inter-class constraints, with no user-defined methods, then an automatically generated *Controller* machine is produced, which INCLUDES all other machines, none of which need to modify each others state and so are only related via USES, which permits (acyclic) dependency structures. An example of this strategy is the B specification generated for the railway signalling system, which is structured as shown in Figure 4.

3.3 Inheritance

Inheritance poses a particular problem in translation of UML to B, since a subclass is dependent on its superclass, and this is an operation dependence (implying the use of INCLUDES in B) since creation of a subclass instance requires invocation of its superclass constructor. However in B, at most one machine can be includes-dependent on a given machine. Thus multiple subclassing of a class in UML cannot be represented using separate machines in B. It is possible to represent linear chains of inheritance using separate machines [22], whereby the

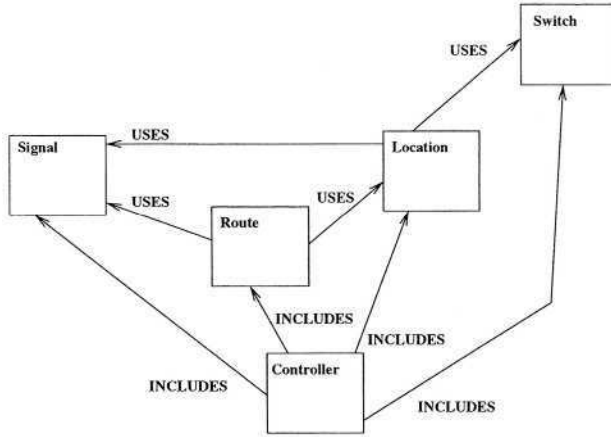


Fig. 4. Structure of Generated B for Signalling System

subclass machine selects the new object to be created and then invokes super-class machine operations to add the instance to their respective instance sets and initialise their local attributes for it, but we prefer to use a consistent approach and always gather together all classes that are related in one inheritance tree and formalise them in a single B machine [9], this is the machine that represents the superclass of all the classes in the tree (Figure 5).

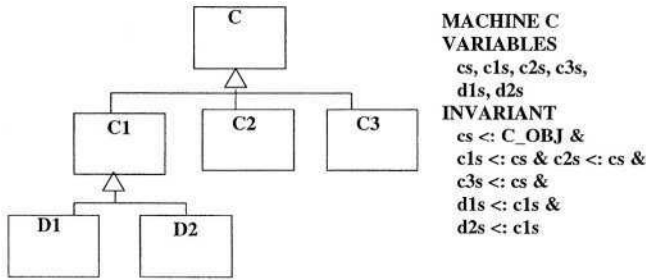


Fig. 5. Representation of Inheritance in B

If class D is an immediate subclass of class C then the axiom $ds \subseteq cs$ is adjoined to the machine M representing them. If D and E are distinct direct subclasses of the same class C then the axiom $ds \cap es = \emptyset$ is also adjoined, and if C is an abstract superclass of a complete set $C_1, \dots, C_n$ of subclasses, then the axiom $c_1s \cup \dots \cup c_ns = cs$ is included. Alternatives, such as *overlapping* subclasses, could also be formalised in the same manner. The B constructor new_D for an immediate subclass D of a class C must select (using ANY) an instance not a member of any class in the hierarchy that includes C and D , and

then add this instance *oox* to each ancestor instance set *es* of *D* as well as to *ds*. It must also set initial values of all the attributes and ONE roles of each ancestor class.

Method polymorphism can be modelled in B as follows. Assume that method $m(p : T) : R$ is defined in class *C* and also given definitions in subclasses *C*₁, *C*₂, ..., *C*_n, then the B definition of *m* is an IF statement in which the case of a more specific class is considered before the case of its immediate superclass, etc. For example if *C*₁ and *C*₂ are subclasses of *C*₃, and *C*₁ and *C*₃ define *m*, the B interpretation would be:

```
mresult <-- m(cx,p) =
  PRE cx: cs & p: T & PreCm
  THEN
    IF cx: c1s THEN DefC1(cx,p)
    ELSE IF cx: c3s THEN DefC3(cx,p)
    ELSE DefC(cx,p) END END
  END
```

where *DefD* is the definition of *m* in *D*. The precondition of *m* in *C* should imply the preconditions of *m* in any subclass, so these are omitted.

3.4 Translation of Constraints

There are three separate categories of constraint:

- *local class constraints* – constraints attached to a single class and involving only features of that class. These are translated into invariants of the B machine representing the class, and update code to preserve these invariants is added within the operations of the machine.
- *non-local class constraints* – other constraints attached to a single class. These are translated into invariants of the *Controller* B machine, and update code to preserve these invariants, written within the operations of the *Controller*.
- *association constraints* – constraints attached to one or more associations. These are also translated into invariants and code in the *Controller*.

Table 3 shows the correspondence of some expressions between OCL, LOCA and B. Sequences (such as ordered associations) and operators such as *sum* also have direct equivalents in B. In general for every OCL expression that is in the LOCA subset (wrt change of syntax) there is a direct corresponding B expression.

There are a number of standard UML constraints on model elements [14]: *frozen* (same as *final* in Java), *classifier* scope (same as *static* in Java), *unique* for attributes, *leaf* (same as *final* in Java) and *abstract* for classes, and *classifier* scope, *abstract* and *query* for operations. Associations may also be *frozen*.

Frozen attributes have no *set* operation, otherwise their interpretation in B is as for other attributes. Frozen associations must be initialised in the constructor, regardless of their cardinality, and have no *set* or other update operations.

Table 3. Correspondence of Expressions from OCL to B

OCL	LOCA	B
$r1.max(r2)$	$\{ r1, r2 \}.max$	$max(\{ r1, r2 \})$
$set \rightarrow union(set2)$	$set \vee set2$	$set \vee set2$
$set \rightarrow intersection(set2)$	$set \wedge set2$	$set \wedge set2$
$set \rightarrow includes(elem)$	$elem : set$	$elem : set$
$set \rightarrow including(elem)$	$set \vee \{ elem \}$	$set \vee \{ elem \}$
$set \rightarrow excludes(elem)$	$elem /: set$	$elem /: set$
$set \rightarrow excluding(elem)$	$set - \{ elem \}$	$set - \{ elem \}$
$set \rightarrow isEmpty()$	$set = \{ \}$	$set = \{ \}$
$set \rightarrow select(P)$	$set (P)$	$\{ x \mid x : set \ \& \ P(x) \}$
$C.allInstances()$	C	cs

Classifier scope attributes $att : T$ are represented as variables $att : T$ in B, not as functions. Unique attributes of class C are represented using an injective function type $att : cs \mapsto T$, and an extra precondition $val \notin att[cs - \{cx\}]$ is required for $setatt(cx, val)$.

Leaf classes cannot have subclasses. Abstract classes have extent the union of the extents of their direct subclasses.

Classifier scope operations omit the object reference parameter ex . Query operations can only modify their *result* variable, and they cannot use the *@pre* pre-state qualifier in their specification. Abstract operations must be specified in each concrete subclass of the abstract class in which they are declared.

3.5 Translation of Class Invariants

Constraints attached to a class C in UML are interpreted as machine invariants of the machine $\mathcal{B}(C)$. The translation makes explicit the OCL ‘flattening’ of sets but otherwise is direct. Table 4 shows the translation of attributes and roles. e' denotes the interpretation of expression e into B. Operators $:$, $=$, etc in LOCA map to corresponding operators in B. However if one side of an equality $a = b$ is set-valued and the other is not, the non-set, say b , must be expressed as $\{b\}$ in the translation. Similarly for other operators ($x : y$ becomes $x \subseteq y$ if x is set-valued, etc).

Each free variable v occurring in a LOCA formula F must have a determinable type T that can be established from the formula. The result of the B translation then universally quantifies v over the translated formula BF :

$$!v. (v : T \Rightarrow BF)$$

Finally the formula is universally quantified by an object cx of the class concerned.

An example of this translation in the case study is from the constraint

`locn = occupied => in.leadSignal.sigset = stop`

to the B invariant

Table 4. Interpretation of Navigation Expressions in B

<i>LOCA</i>	<i>B</i>
Variable or constant x , primitive or string value x	x
Attribute of single-valued expression $obj.att$	$att(obj')$
Attribute of set-valued expression $s.att$	$att[s']$
Role of single-valued expression $obj.role$	$role(obj')$
Cardinality ONE role of set-valued expression $s.role$	$role[s']$
Non-ONE role of set-valued expression $s.role$	$union(role[s'])$

```
!locationx.(locationx: locations =>
  (locn(locationx) = occupied =>
    sigset[leadSignal[in(locationx)]] = {stop}))
```

If there was a many role *in* from *Location* to *Route*.

The B translation of class invariants are placed in the machine representing the class *provided* they only refer to local features of that class. Other invariants are made the responsibility of the system controller.

The constructor *new_C* in $\mathcal{B}(C)$ must check that the local class invariants hold for the initial values to be assigned to the attributes and roles. Each *set* operation must also do this for the new values assigned to the attribute or role. Code is generated to preserve local invariants involving two or more non-frozen attributes, eg: the operation *setready(obj, readyx)* must also set *traversable(obj) := false* if *readyx = false*.

3.6 Translation of Inter-class Constraints

Constraints may be attached to several associations, and therefore also implicitly to the classes related by these associations. In the translation to B, we invent a variable *cx* for each class *C* in the context of the constraint, and universally quantify over these variables. Table 5 shows the quantifier range set of the *cx* variables ranging over the instances of each class in each case of an association $E1^{role1} \rightarrow^{role2} E2$. Then in the quantified formula *F* we interpret a reference *feat* to an attribute or role of class *E* by *feat(vare)* where *vare* is the quantified variable created for *E*. The rules given in Table 4 are also applied if *F* involves navigation expressions.

As an example of the mapping, the constraint

```
locn = occupied => rte = occupied
```

becomes the B invariant

```
!locationx.(locationx: locations =>
  !routex.(routex: routes & locationx: path(routex) =>
    (locn(locationx) = occupied =>
      rte(routex) = occupied)))
```

of the *Controller* machine for the signalling system.

Table 5. Mapping of Inter-class Constraints to B Quantifier Restrictions

<i>LOCA</i>	<i>B</i>
Class 2 at role 2 (ONE) end of association where class 1 quantifier is already known, on <i>var1</i>	$var2 = role2(var1)$
Class 2 at role 2 (non-ONE) end of association where class 1 quantifier is known, on <i>var1</i>	$var2 : role2(var1)$
Class 1 at role 1 end of association where class 2 quantifier is known, on <i>var2</i> , <i>role2</i> is ONE	$var2 = role2(var1)$
Class 1 at role 1 end of association where class 2 quantifier is known, on <i>var2</i> , <i>role2</i> non-ONE	$var2 : role2(var1)$

3.7 Synthesis of Event Code

An invariant such as

$$locn = occupied \Rightarrow rte = occupied$$

can be interpreted as an instruction to set *rte* to *occupied* for each *Route* object related to a *Location* object *loc* whenever *loc*'s *locn* is set to *occupied*, ie, as a description of the necessary actions to be taken to preserve the truth of the invariant.

In general, for each constraint *C*, response code for an event α will need to be generated (to maintain *C*) if

$$Pre_{\alpha} \wedge C \Rightarrow [\alpha]C$$

is not true, where $[\alpha]C$ denotes the weakest precondition of *C* with respect to α , and Pre_{α} is the specified precondition of α . The response code will modify a designated *write frame*: for local invariants this is the set of non-frozen data features of the class, excluding the feature directly modified by α . If it is impossible to produce such code, the precondition must be strengthened instead.

In the case of a local invariant of a class, the response code will be expressed using assignments $:=$ and no operation calls, within the code of α in the B machine representing the class. For non-local constraints the code synthesis process generates the updates required on all sets of objects related to a particular object via a constraint. If object *oo* of class *C* changes state via an event *attval(oo)* of the system, for example, then there will be a set *dset* of objects of a related class *D* affected, and the constraint will prescribe how their states should change.

Therefore the translation derives the set *dset* of affected objects for each class *D* related to the event origin class *C* via the associations of the constraint. Table 6 shows the set of affected objects of each class in each situation.

The constraint conclusion is translated into B using these sets as the arguments of suitable *setAllatt(objectset, val)*, *setatt(obj, val)*, etc, operations. A feature *feat* of class *C* is interpreted as *feat[cset]* where *cset* is the set of affected objects of *C*, or as *union(feat[cset])* if *feat* is a non-ONE cardinality role.

Table 6. Sets of Objects Affected by an Event

<i>LOCA</i>	<i>B</i>
Class 2 at role 2 (ONE) end of association; class 1 set of objects affected is <i>set1</i>	$role2[set1]$
Class 2 at role 2 (non-ONE) end; class 1 set of objects affected is <i>set1</i>	$union(role2[set1])$
Class 1 at role 1 end of association; <i>set2</i> are class 2 objects affected, <i>role2</i> is ONE	$\{c1x \mid c1x : c1s \wedge role2(c1x) : set2\}$
Class 1 at role 1 end of association; class 2 objects affected are <i>set2</i> , <i>role2</i> non-ONE	$\{c1x \mid c1x : c1s \wedge role2(c1x) \cap set2 \neq \emptyset\}$

Table 7. Simplification of B Expressions

<i>B Expression</i>	<i>Rewritten Expression</i>
$f[\{x\}]$	$\{f(x)\}$
$union(\{x\})$	x
$union(\emptyset)$	$\emptyset$
$v \cap \{y\} \neq \emptyset$	$y : v$
$y : \{x\}$	$y = x$
$\{x\} = \{y\}$	$x = y$
$\{x\} \subseteq y$	$x : y$
$setAllfeat(\{x\}, val)$	$setfeat(x, val)$
$addAllrole(\{x\}, obj)$	$addrole(x, obj)$
$removeAllrole(\{x\}, obj)$	$removerole(x, obj)$

A number of simplification rules are applied to reduce the complexity of the resulting formula (Table 7).

Applying these rules, the constraint

$$locn = occupied \Rightarrow rte = occupied$$

together with the transitive composition of *C1* and *C7* gives the following code for the controller operation *locnoccupied(oo)*:

```

locnoccupied(oo) =
  PRE oo: locations
  THEN
    setlocn(oo,occupied) ||
    setAllrte({ routex |
               routex: routes & oo: path(routex) }, occupied) ||
    setAllsigset(
      leadSignal[{ routex |
                  routex: routes & oo: path(routex) }], stop)
  END

```

All updates which result from a given sensor event must be performed together (using || and IF statements) in the same controller operation.

The *Controller* machine is responsible for maintaining inter-class invariants and coordinating the response to input events. Its typical form is:

```

MACHINE Controller
SEES SystemTypes
INCLUDES C1, ..., Cn /* All machines generated for classes */
PROMOTES new_C1, ...
INVARIANTS /* All inter-class invariants */
OPERATIONS
  attvalue(oo) =
    PRE oo: cs
    THEN
      setatt(oo,val) || synthesised code
    END;
  ....
END

```

There is an operation *attval* for each sensor attribute *att* of each class, and each value *val* (for *att* of enumerated type) of its type.

3.8 Operation Specifications

In OCL and LOCA operation behaviour can be specified using pre/post constraints:

```

context E::op(p: T): R
pre: P
post: Q

```

In *Q* the initial values of variables at start of the operation are referred to as *var@pre*.

Provided only local features are updated, such an operation specification can be translated into a B operation

```

resultx <-- op(ex,p) =
  PRE p: T' & ex: es & P(ex)
  THEN
    ANY v,result
    WHERE Q[x/x@pre,v/x](ex) & v: Tx & result: R'
  THEN
    x(ex) := v || resultx := result
  END
END

```

where $P(ex)$ denotes the predicate P with each feature f of E replaced by $f(ex)$, feature application $obj.F$ expressed as $F(obj)$, $F[obj]$ or $union(F[obj])$ as appropriate, and there is a new variable v_f for each feature f of E , the substitution of $x@pre$ by x is carried out for each $x@pre$ in Q . B can then be used to check that the operation specification is consistent with (preserves) the

local class invariant. If several classes in the same super/subclass heirarchy also define *op*, with identical parameter type *T*, then the modelling of polymorphism described in Section 3.3 can be used, where *DefE* is the *ANY* statement.

As an example, the specification

```
context E::op()
pre: x < y
post: x > x@pre & y > x
```

where *x* and *y* are *int*-valued attributes of *E*, becomes:

```
op(ex) =
  PRE ex : es & x(ex) < y(ex)
  THEN
    ANY vx,vy
    WHERE vx > x & vy > y & vx : INT & vy : INT
    THEN
      x(ex) := vx || y(ex) := vy
    END
  END
```

This can be shown to preserve $!ex.(ex : es \Rightarrow x(ex) < y(ex))$.

A *query* operation only modifies *result* and has a simpler translation.

In UML-RSDS, action invariants $P \Rightarrow AX(Q)$ are an alternative means to specify state changes. An OCL pre/post constraint can be expressed as an action invariant

$$v = x \ \& \ P \Rightarrow AX(Q[v/x@pre])$$

where the *v* axe new variables, one for each *x* such that *x@pre* occurs in *Q*.

4 Specification Verification and Validation

The UML-RSDS tool performs basic consistency and completeness checks on sets of constraints. In addition the translation to B can be used to proof check the specification and identify other errors. This is because the translation has itself been verified, that is, the translation into B of a UML-RSDS model *M* can be shown to validate each of the axioms of the semantics Γ_M of *M* [13]. This means that if a property can be derived in *M*, it will also be true in the B translation. In particular, contradictions will be detectable, in principle, in B.

Incompleteness of the UML-RSDS model can also be detected via animation of the translated B, enabling particular scenarios to be symbolically executed. For example in the railway specification the lack of an invariant to reset a *route* to unoccupied if all its locations are unoccupied can be identified in this way (the required invariant is $path.locn = unoccupied \Rightarrow rte = unoccupied$). B can also be used to check if there are non-vacuous models of a specification.

As an example of property verification using B, it is easy to write UML class diagrams which are valid according to the UML definitions, but which have only

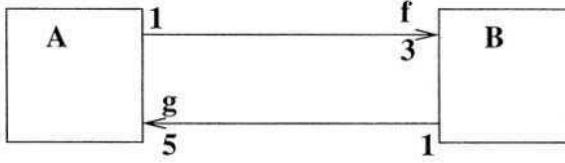


Fig. 6. Vacuous UML Model

vacuous implementations. In Figure 6 there are two classes with associations to each other. Superficially this seems sensible, however the formal semantics of this situation, expressed by the translation to B, is:

$$\begin{aligned}
 f &: as \rightarrow \mathbb{F}(bs) \\
 g &: bs \rightarrow \mathbb{F}(as) \\
 \forall a.(a : as \Rightarrow \text{card}(f(a)) = 3) \\
 \forall b.(b : bs \Rightarrow \text{card}(g(b)) = 5) \\
 \text{union}(f[as]) &= bs \\
 \text{union}(g[bs]) &= as \\
 \forall a, a'.(a : as \wedge a' : as \wedge a \neq a' \Rightarrow f(a) \cap f(a') = \emptyset) \\
 \forall b, b'.(b : bs \wedge b' : bs \wedge b \neq b' \Rightarrow g(b) \cap g(b') = \emptyset)
 \end{aligned}$$

This means that the $f(a)$ form a partition of bs , indexed by elements of as , so $\text{card}(bs) = 3 * \text{card}(as)$, and similarly $\text{card}(as) = 5 * \text{card}(bs)$. This is only possible if $as = bs = \emptyset$.

The consistency of statechart specifications can also be checked against class diagram invariants: statechart transitions can possess generated actions, which consist of method invocations on supplier objects. For example, a route object could have the generation *leadSignal.setsigset(stop)* on transitions for *setrte(occupied)*. These explicit commands are asserted to *implement* certain constraints (*C7* in this case) and their translation into B replaces the B code synthesised from the declarative constraints that they are claimed to implement. Consistency checking the controller machine or machines derived from entities (for local constraints and implementing actions) can be used to verify that the constraints are indeed ensured by the statechart actions.

5 Translations to Java and SMV

A similar translation approach can be used to define an executable implementation of a system in Java [12], and a description in SMV [4] which can be used for automated temporal verification (model checking). The translation into Java is simpler than that to B because of the closer structural relation between Java and UML, and the ability in Java to perform complex updates as a series of individual object updates, whereas in B they must be performed in a single step. Details of the Java translation are given in [12].

The UML-RSDS, B and SMV semantics are very closely related, representing a model or execution of a specification as a sequence of time steps, each step

consisting of no events or of a single sensor event and a set of other events (the reaction to the sensor event). Thus the results of analysis in B or SMV can be immediately related to the UML model.

6 Comparison

Related work on UML is the U2B tool of Butler [19], and translations [7] from UML to Object-Z [18]. These translations emphasise the preservation of the UML class diagram structure in the formal notation, as the starting point of a formal development. In both cases the formal notation is central and the UML used only as a diagrammatic front end. In contrast, we envisage that most developers would prefer to use UML as their main notation, and only use B as one analysis tool for property checking and animation of the class diagram. The exact structure of formal modules is therefore less important than the traceability of formal notation elements to the original diagram elements. We have ensured such a traceability is possible by representing each class, attribute and role by a separate variable and each method by a B operation.

Other related tools are: (i) The KeY System. This toolkit [2] provides facilities for verifying object-oriented applications against OCL specifications. In contrast to KeY, UML-RSDS is intended to be used by developers as an OO formal method, and to generate applications from high-level models according to the MDA [16] concepts. We also rely on established predicate logic as the basis of verification (in B) instead of a new dynamic logic. RSDS has also established a modular design methodology including modular verification, which is lacking in KeY.

(ii) The USE Tool. This tool [17] provides validation and verification of UML specifications by means of checking test case object diagrams against the specification. Similar capabilities are provided by UML-RSDS via translation to B, which supports animation and monitoring of preconditions and invariants. Translation to SMV provides additional capabilities of temporal property verification, and identification of sequences of behaviour which are counter-examples to system invariants.

(iii) Alloy. The Alloy constraint language and tools [6] are also intended to provide a lightweight specification notation and semantic error detection capabilities for it. However the UML-RSDS constraint language is closer to OCL than is Alloy, and Alloy retains explicit quantifiers and set comprehension, so requiring a greater level of mathematical experience to use for specification than UML-RSDS. Our analysis approach is based on using established formal methods tools (B and SMV) instead of developing a new tool.

UML-RSDS is similar in intent to pragmatic formal approaches such as SCR [3]. However, instead of inventing a new diagrammatic formal method, we make precise an existing widely-used semi-formal method.

Conclusion

We have shown that a simplified form of OCL can be used for practical formal specification of reactive systems, and that a translation of UML class diagrams to B can be performed together with synthesis of B operation code from constraints.

References

1. J-R. Abrial, *The B Method*, Cambridge University Press, 1996.
2. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hahnle, W. Menzel, and P. H. Schmitt. *The KeY approach: Integrating object oriented design and formal verification*. Technical Report 2000/4, University of Karlsruhe, Department of Computer Science, Jan. 2000.
3. R. Bharadwaj, C. Heitmeyer, *Model checking complete requirements specifications using abstraction*. In *Proceedings of Automated Software Engineering*, 6, 37–68, 1999.
4. J R Burch, E M Clarke, K L McMillan, D L Dill, J Hwang, *Symbolic Model Checking: 10²⁰ States and Beyond*, Proceedings of the Fifth Annual Symposium on Logic in Computer Science, 1990.
5. CS-RR Inc., *CS-RR Software User Requirements Document*, 1994.
6. Daniel Jackson, *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*, Software Design Group, MIT Lab for Computer Science, 2002.
7. S. Kim, D. Carrington, *A Formal Mapping Between UML Models and Object-Z Specifications*, in ZB2000, LNCS Vol. 1878, Springer-Verlag, 2000.
8. K. Lano, D. Clark, K. Androutsopoulos, *Safety and Security Analysis of Object-oriented Models*, Safecomp 2002.
9. K. Lano, H. Haughton, *Specification in B*, Imperial College Press, 1996.
10. K. Lano, J. Fiadeiro, L. Andrade, *Software Design in Java 2*, Palgrave, 2002.
11. K. Lano, D. Clark, K. Androutsopoulos, *Formal Specification and Verification of Railway Systems using UML*, FORMS 2003.
12. K. Lano, D. Clark, K. Androutsopoulos, *Synthesis of Code from UML Specifications*, DCS, King's College, 2003.
13. K. Lano, D. Clark, K. Androutsopoulos, *Extended Axiomatic Semantics of UML Class Diagrams and Statecharts*, DCS, King's College, 2003.
14. OMG, *UML Version 1.5 Specification*, <http://www.omg.org/uml/>, 2003.
15. OMG, *Response to UML 2.0 OCL RfP*, OMG Document ad/2003-01-07, 2003.
16. OMG, *Model-Driven Architecture*, <http://www.omg.org/mda/>, 2003.
17. Richters, M. A *UML-based Specification Environment*, <http://www.db.informatik.uni-bremen.de/projects/USE>, 2001.
18. G. Smith, *The Object-Z Specification Language*, Kluwer, 2000.
19. C. Snook, P. Wheeler, M. Butler, *Preliminary Tool Extensions for Integration of UML and B*, IST-2000-30103 deliverable D4.1.2, 2003.
20. H. Treharne, *Supplementing a UML Development Process with B*, FME '02.
21. J. Warmer, A. Kleppe, *The Object Constraint Language: Precise Modelling with UML*, Addison-Wesley, 1999.
22. P. Zeppo, *From UML to B Specifications*, MSc thesis, Dept. of Computer Science, King's College London, 2002.

Software Verification with Integrated Data Type Refinement for Integer Arithmetic

Bernhard Beckert¹ and Steffen Schlager²

¹ University of Koblenz-Landau, Institute for Computer Science
D-56072 Koblenz, Germany
beckert@uni-koblenz.de

² University of Karlsruhe, Institute for Logic, Complexity and Deduction Systems
D-76128 Karlsruhe, Germany
schlager@ira.uka.de

Abstract. We present an approach to integrating the refinement relation between infinite integer types (used in specification languages) and finite integer types (used in programming languages) into software verification calculi. Since integer types in programming languages have finite ranges, in general they are not a correct data refinement of the mathematical integers usually used in specification languages. Ensuring the correctness of such a refinement requires generating and verifying additional proof obligations. We tackle this problem considering JAVA and UML/OCL as example. We present a sequent calculus for JAVA integer arithmetic with integrated generation of refinement proof obligations. Thus, there is no explicit refinement relation, such that the arising complications remain (as far as possible) hidden from the user. Our approach has been implemented as part of the KeY system.

Keywords: Software verification, specification, UML/OCL, data refinement, Java, integer arithmetic.

1 Introduction

The Problem. Almost all specification languages offer infinite data types, which are not available in programming languages. In particular this holds for the mathematical *integer* data type which we will focus on in this paper. Infiniteness of integer types on the specification level is an important feature of a specification language for two reasons:¹

1. Specifications should be abstract and independent of a concrete implementation language.
2. Developers think in terms of arithmetic on integers of unrestricted size.

¹ For these reasons, Chalin [7] proposes to extend the JAVA Modeling Language (JML) [18], which does not support infinite integer types, with a type `infint` with infinite range.

In the implementation, the infinite types have to be replaced with finite data types offered by the programming language. Verifying the correctness of the implementation requires among other things to show that this replacement does not cause problems. Speaking in terms of refinement one has to prove that the finite types are a correct *data refinement* of the specification language types (in the particular context where they are used). This is done by generating additional proof obligations for each arithmetical expression stating that the result does not exceed the finite range of the type of the expression. By verifying these additional proof obligations, we establish that the programming language types are only used to the extent that they indeed are a refinement of the specification language types. This check cannot be done once and for all but has to be repeated for each particular program. It is tedious and error-prone if done by hand.

Our Solution. Our solution to the integer data refinement problem is to define a verification calculus that combines the infinite integer semantics of specification languages and the finite integer semantics of programming languages. To avoid “incidentally” correct programs (as defined below), we verify that no overflow² occurs during the execution of a program, i.e., a pre-condition is added to each arithmetical operation stating that its result is within the bounds of the JAVA data type. That is, we are not content with merely showing that a program satisfies its specification, which it may do even if an overflow occurs.

To keep all these complications hidden from the user as far as possible, the relation between the different types of integer semantics is not made explicit (there is no formal refinement relation). Instead, the handling of the refinement relation and, in particular, the generation of proof obligations to make sure that no overflow occurs, is integrated into the rules of the verification calculus.

Our Choice of Specification and Implementation Language. In this paper, we use JAVA as implementation language, and the specification language we consider is UML/OCL.

Note, however, that our particular choice of specification and implementation languages is not crucial to our approach. The languages UML/OCL and JAVA can be substituted by almost any other specification and implementation languages (e.g. Z [22] or B [1] resp. C++). We use UML/OCL and JAVA mainly because the work presented here has been carried out as part of the KeY project [2,3] (see <http://www.key-project.org>). The goal of KeY is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. We decided to use UML/OCL as specification language since the Unified Modeling Language (UML) [19] has been widely accepted as

² The situation that the result of an arithmetical operation exceeds the maximum or minimum value of its type is called *overflow*. In JAVA, if overflow occurs the result is computed modulo the size of the data type. For example, $\text{MAX\_int} + 1 = \text{MIN\_int}$ (where MAX_int and MIN_int are the maximum resp. minimum value representable in type `int`).

the standard object-oriented modelling language and is supported by a great number of CASE tools. The programs that are verified should be written in a “real” object-oriented programming language. We decided to use JAVA (actually KeY only supports the subset JAVA CARD, but the difference is not relevant for the topic of this paper).

Motivation for Our Solution. The motivation for our solution is that using the semantics of JAVA (as implemented by a JAVA Virtual Machine) to verify that a program correctly implements its specification (without checking for overflow) may still lead to undesired results if the specification is too weak. A formally correct program may not reflect the intentions of the programmer if overflow occurs during its execution—even if its observable behaviour satisfies the specification. Such programs, which we call “incidentally” correct, are a source of error in the software development process (as explained in Section 2.2). The problem is aggravated by the fact that JAVA, as well as many other programming languages like C++ and Pascal, do not indicate overflow in any way (in some other languages, such as Ada, an exception is thrown). Moreover, many JAVA programmers are not aware of this behaviour of JAVA integers.³ But even programmers who know about this JAVA feature make errors related to overflow. For example, in [6] a flaw arising from unintended overflow in the implementation of Gemplus’ electronic purse case study [17] is discovered. The result of this flaw is that the method `round` which is supposed to return the closest integer, in fact returns `-32768` when invoked with `32767.999`.

Dynamic Logic. For the verification component in the KeY system, we use an instance of Dynamic Logic. This instance, called JavaDL, can be used to specify and reason about properties of JAVA CARD programs [4].

Dynamic Logic (DL) is a modal predicate logic with a modality $\langle p \rangle$ for every program p (we allow p to be any sequence of legal JAVA statements); $\langle p \rangle$ refers to the successor worlds (called states in the DL framework) that are reachable by running the program p . In standard DL there can be several of these states (worlds) because the programs can be non-deterministic; but here, since JAVA programs without threads are deterministic (so far concurrency is not considered in the KeY system), there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). The formula $\langle p \rangle \phi$ expresses that the program p terminates in a state in which ϕ holds. A formula $\psi \rightarrow \langle p \rangle \phi$ is valid if for every state s satisfying the pre-condition ψ , a run of the program p starting in s terminates, and in the terminating state the post-condition ϕ holds.

To prove the correctness of a program, one has to prove the validity of DL formulas (*proof obligations*) that are generated from the UML/OCL specification and the JAVA implementation. The approach for generating proof obligations used in the KeY project is described in [5].

³ The claim that many programmes are not aware of the behaviour of JAVA integers in case of an overflow is based on the authors’ personal experiences made in teaching courses for computer science students and conversations with programmers working in industry.

Deduction in DL is based on symbolic program execution and simple program transformations and is, thus, close to a programmer's understanding of JAVA.

Related Work. So far, research in this area mainly focused on formalising and verifying properties of floating-point arithmetic [12,13] (following the IEEE 754 standard). However, there are good reasons not to neglect integer arithmetic and in particular integer arithmetic on finite programming language data types. For example, integer overflow was involved in the notorious Ariane 501 rocket self-destruction, which resulted from converting a 64-bit floating-point number into a 16-bit signed integer. To avoid such accidents in the future the ESA inquiry report [9] explicitly recommended to “verify the range of values taken by any internal or communication variables in the software.”

Approaches to the verification of JAVA programs that take the finiteness of JAVA's integer types into consideration—but not their relationship to the infinite integer types in specification languages—have been presented in [16,23].

The verification techniques described in [20,25,15] treat Java's integer types as if they were infinite, i.e., the overflow problem is ignored.

In [10], a problem in the JAVA CARD language specification is pointed out. Certain JAVA CARD programs containing integer computations with the unsigned shift-operator `>>>` give different results on the JAVA resp. the JAVA CARD platform.

Closely related to our approach is Chalin's work [7]. He argues that the semantics of JML's arithmetic types (which are finite as in JAVA) diverges from the user's intuition. In fact, a high number of published JML specifications are shown to be inadequate due to that problem. As a solution, Chalin proposes an extension of JML with an infinite arithmetic type.

Structure of this Paper. The structure of this papers is as follows: After explaining in Section 2 why using only one of the two integer semantics (infinite as in UML/OCL resp. finite as in JAVA) is problematic, we explain our approach that is based on combining both semantics in Section 3. In Section 4, we describe the sequent calculus for the combined semantics, which has been implemented in the KeY system. Finally, in Section 5 we give an example for using our approach in software development.

Due to space restrictions, the proofs of the theorems given in the following are only sketched, they can be found in [21].

2 Disadvantages of Not Combining Finite and Infinite Integer Types

In this section we explain the mutual deficiencies of the two integer semantics when used separately.

2.1 Disadvantages of Using an Infinite Integer Type

A concrete implementation can be regarded as a refinement of a given specification where, in particular, the data types used in the specification are refined by concrete data types available in the implementation language. Following [14], we say that a (concrete) data type correctly refines an (abstract) data type if in all circumstances and for all purposes the concrete type can be validly used in place of the abstract one.

Considering OCL and JAVA this means that the primitive JAVA type `int` (byte, short, long could be used as well) is used to implement the specification type `INTEGER`. Obviously, this is not a correct data type refinement in general. For example, the formula $\forall x. x + 1 > x$ is valid with x of type `INTEGER` but is not valid if the type `INTEGER` is replaced with `int` because it holds $\text{MAX\_int} + 1 = \text{MIN\_int}$ and thus $\text{MAX\_int} + 1 < \text{MAX\_int}$.

In the following, a semantics for JavaDL that treats JAVA integers as if they were correct refinements of `INTEGER` is called $\mathcal{S}_{OCL}$. This semantics $\mathcal{S}_{OCL}$, where overflow is totally disregarded, allows to verify programs that do not satisfy the specification, which is not just a disadvantage but unacceptable.

2.2 Disadvantages of Using a Finite Integer Type

A semantics that uses finite integer types and exactly corresponds to the semantics defined in the JAVA language specification [11] (and thus to the semantics implemented by the JVM) is called $\mathcal{S}_{Java}$ in the following. Using semantics $\mathcal{S}_{Java}$, the validity of the JavaDL proof obligations implies that all specified properties hold during the execution of the program on the JVM. Thus, at first sight $\mathcal{S}_{Java}$ seems to be the right choice. But there are also some drawbacks which are discussed in the following.

If a program is correct using semantics $\mathcal{S}_{Java}$ it shows the expected verified functional behaviour (black-box behaviour). However, overflow may occur during execution leading to a discrepancy between the developer's intention and the actual internal (white-box) behaviour of the program. As long as neither specification nor implementation are modified this discrepancy has no effect. However, in an ongoing software development process programs are often modified. Then, a wrong understanding of the internal program behaviour easily leads to errors that are hard to find, precisely because the program behaviour is not understood.

For example, using $\mathcal{S}_{Java}$, the formula $i > 0 \rightarrow \{i=i+1; i=i-1;\} i > 0$ is valid although in case the value of i is MAX_int , an overflow occurs and the value of i is (surprisingly) negative in the intermediate state after the first assignment. The program shows the expected black-box behaviour but the white-box behaviour likely differs from the developer's intention.

As mentioned in the introduction, we call such programs that satisfy their specification but lead to (unexpected) overflow during execution "incidentally" correct, because we assume that the white-box behaviour of the program is not understood. In our opinion "incidentally" correct programs should be avoided

because they are a permanent source of error in the ongoing software development process.

The above problem does not arise directly from the semantics $\mathcal{S}_{Java}$ itself but rather from the semantic gap between the specification language UML/OCL and the implementation language JAVA. Thus, the same problem also occurs with other specification and implementation languages.

Another disadvantage of $\mathcal{S}_{Java}$ is the fact that formulas, that are intuitively valid in mathematics like $\forall x.\exists y.y > x$ are not valid anymore if x, y are of a built-in JAVA type, like e.g. `int`.

Furthermore, using semantics $\mathcal{S}_{Java}$ requires to reason about modulo arithmetic. This is more complicated than reasoning about integers because many simplification rules known from integer arithmetic cannot be applied to modulo arithmetic (for example, in modulo arithmetic $x + 1 > x$ cannot be simplified to *true*).

Our experience shows that many proof goals involving integer arithmetics (that remain after the rules of our JavaDL calculus have been applied to handle the program part of a proof obligation) can be discharged automatically by decision procedures for arithmetical formulas. In the KeY prover we make use of freely available implementations of arithmetical decisions procedures, like the Cooperating Validity Checker [24] and the Simplify tool (which is part of ESC/Java [8]). Both do *not* work for modulo arithmetics.

3 Combining Finite and Infinite Integer Types

3.1 The Idea

Basically, there are two possible approaches to proving that a particular JAVA program (with finite integer types) correctly refines a particular UML/OCL specification (with infinite integers types).

Firstly, one can show that the observable behaviour of the program meets the specification (whether overflow occurs or not), without checking explicitly that there is any particular relation between the integer types in the program resp. the specification. This amounts to using semantics $\mathcal{S}_{Java}$, which allows “incidentally” correct programs.

Secondly, one can show that whenever one of the arithmetical operations⁴ $\circ \in \{+_T, -_T, *_T, /_T, \%_T\}$ on a type⁵ $T \in \{\text{int}, \text{long}\}$ is invoked during the execution of a program, the following pre-condition is met, ensuring that no overflow occurs:

⁴ Here, we do not consider the bit-wise logical and shift operations on integers, i.e., $\sim$ (complement), $\&$ (and), $\mid$ (or), $\wedge$ (xor), $\ll$ (left shift), $\gg$ (right shift), $\ggg$ (unsigned right shift). They may cause an overflow effect, but a programmer using bit-wise logical or shift operators can be assumed to be aware of the data type’s bit-representation and, thus, of its finiteness.

⁵ In JAVA arithmetical operators exist only for the types `int` and `long`. Arguments of type `byte` or `short` are automatically cast to `int` (or to `long` if one operand is of type `long`) by the JVM. This is called *promotion*.

$$\text{MIN\_}T \leq x \hat{o} y \leq \text{MAX\_}T,$$

where $\hat{o}$ is the UML/OCL operation on INTEGER corresponding to the JAVA operation o . By checking this pre-condition, we establish that the JAVA types are only used to the extent that they indeed are a refinement of the UML/OCL types. This check cannot be done once and for all but has to be repeated for each particular JAVA program.

We use this second approach that truly combines the two types of integer semantics and avoids “incidentally” correct programs. The generation of proof obligations corresponding to instances of the above pre-condition is built into our verification calculus (Section 4).

With our approach to handling JAVA’s integers, we fulfil the following three demands:

1. If the proof obligation for the correctness of a program is discharged, then the program indeed satisfies the specification. That is, the semantics of JavaDL and our calculus correctly reflect the actual JAVA semantics.
2. Programs that are merely “incidentally” correct (due to unintended overflow) *cannot* be proved to be correct, i.e., the problem is detected during verification.
3. Formulas like $\forall x.\exists y.y > x$ that are valid over the (infinite) integers (and, thus, are valid according to the user’s intuition) remain valid in our logic.

3.2 A More Formal View

This section gives a formal definition of our semantics $\mathcal{S}_{\text{KEY}}$ for the JAVA integers that combines the advantages of (finite) JAVA and (infinite) UML/OCL integer semantics.

We extend JAVA by the additional primitive data types

`arithByte`, `arithShort`, `arithInt`, `arithLong`,

which are called *arithmetical types* in contrast to the built-in types `byte`, `short`, `int`, and `long`. The new arithmetical types have an *infinite* range. They are, however, not identical to the mathematical integers (as used in $\mathcal{S}_{\text{OCL}}$) because the semantics of their operators in case of an “overflow” is different (in fact, it remains unspecified).

Note, that this extension of JAVA syntax is harmless and does not require an adaptation of the JAVA compiler. The additional types are only used during verification. Once a program is proved correct, they can be replaced with the corresponding built-in types (Corollary 1 in Section 3.3).

Definition 1. *Let p be a program containing arithmetical types. Then the program $p\text{transf}(p)$ is the result of replacing in p all occurrences of arithmetical types with the corresponding built-in JAVA types.*

Theorem 1. *If a JAVA program p is well-typed, then the program $p\text{transf}(p)$ is well-typed.*

An obvious difference between our semantics $\mathcal{S}_{KeY}$ and $\mathcal{S}_{OCL}$ resp. $\mathcal{S}_{Java}$ is that the signatures of the underlying programming languages differ, since $\mathcal{S}_{KeY}$ is a semantics for JAVA with arithmetical types whereas $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$ are semantics for standard JAVA.

Because of their infinite range, not all values of an arithmetical type are representable in the corresponding built-in type. There are program states⁶ in JavaDL with $\mathcal{S}_{KeY}$ that do not correspond to any state reachable by the JVM. In the following, we call such states “unreal”.

Definition 2. *A variable or an attribute that has an arithmetical type T is in valid range (in a certain state) iff its value val satisfies the inequations*

$$\text{MIN\_}T' \leq val \quad \text{and} \quad val \leq \text{MAX\_}T' ,$$

where T' is the built-in JAVA type corresponding to T .

Definition 3. *A JavaDL state s is called a real state iff all program variables and attributes with an arithmetical type are in valid range. Otherwise, s is called an unreal state.*

As already mentioned, both $\mathcal{S}_{KeY}$ and $\mathcal{S}_{OCL}$ have the same infinite domain. The crucial difference is in the semantics of the operators: If the values of the arguments of an operator application in $\mathcal{S}_{KeY}$ are in valid range but the (mathematical) result is not (i.e., overflow would occur if the arithmetical types were replaced with the corresponding built-in types), then the result of the operation is unknown; it remains unspecified. Otherwise, i.e., if the result is in valid range, it is the same as in $\mathcal{S}_{OCL}$. Technically this is achieved by defining that the result is calculated in the overflow case by invoking a method `overflow(x,y,op)` (the third parameter `op` is the operator that caused overflow and `x,y` are the arguments), whose behaviour remains unspecified (it does not even have to terminate).

The method `overflow` is not invoked if at least one argument of the operation is already out of valid range. In that case, the semantics of the operation in our semantics $\mathcal{S}_{KeY}$ is the same as in $\mathcal{S}_{OCL}$. This definition cannot lead to incorrect program behaviour because the program state before executing the operation is unreal and cannot be reached in an actual execution of the program.

The main reason for leaving the result of integer operations unspecified in the overflow case is that no good semantics for the overflow case exists, i.e., there is no reasonable implementation for the method `overflow`. In particular, the following two implementations that seem useful at first have major drawbacks:

- The method `overflow` throws an exception, does not terminate, or shows some other sort of “exceptional” behaviour. Then the semantics differs from the actual Java semantics (where an overflow occurs without an exception

⁶ A program state assigns values (of the appropriate type) to local program variables, static fields, and the fields of all existing objects and arrays.

Table 1. Comparison of properties of $\mathcal{S}_{OCL}$, $\mathcal{S}_{Java}$, and $\mathcal{S}_{KeY}$.

Property	$\mathcal{S}_{OCL}$	$\mathcal{S}_{Java}$	$\mathcal{S}_{KeY}$
Underlying programming language	JAVA	JAVA	extended JAVA
Overflow on built-in integer types	no	yes	yes
Overflow on arithmetical types	—	—	no
Range of built-in integer types	infinite	finite	finite
Range of arithmetical types	—	—	infinite
Existence of unreal states	yes	no	yes
Behaviour of programs in DL and on the JVM	different	equal	equal under certain conditions

being thrown). This leads to the same problem as with semantics $\mathcal{S}_{OCL}$, i.e., programs whose actual execution does not satisfy the specification could be verified to be correct.

- The method `overflow` calculates the result in the same way as it is done in JAVA, including overflow. This leads to the same problem as with semantics $\mathcal{S}_{Java}$, i.e., “incidentally” correct programs could be verified to be correct.

The instance of $\mathcal{S}_{KeY}$ that results from using the latter of the above two implementations for `overflow` (instead of leaving it unspecified) is very similar to $\mathcal{S}_{Java}$. In the following, it is therefore called $\mathcal{S}_{Java'}$. While the problem that programs may be only “incidentally” correct remains with $\mathcal{S}_{Java'}$, it has an advantage over $\mathcal{S}_{Java}$: Using arithmetical types, formulas like $\forall x.\exists y.y > x$ are valid (other differences are discussed later).

Another reason for leaving `overflow` unspecified is that, if a JavaDL formula ϕ is derivable in our calculus for JavaDL based on $\mathcal{S}_{KeY}$ (i.e. `overflow` remains unspecified), then ϕ is valid for *all* implementations of `overflow` (this follows from the soundness of the calculus). In particular, one can conclude that (1) ϕ is valid in semantics $\mathcal{S}_{Java'}$ and (2) the validity of ϕ is not “incidental” (due to an `overflow`).

Example 1. The formula

$$\langle j=i+1; \rangle j \doteq i+1$$

(where i, j are of an arithmetical type T) is *not* valid and not provable in our calculus (because $j=i+1$ may cause an `overflow` after which $j = i+1$ does not hold).

However, the formula

$$i > \text{MAX\_T} \rightarrow \langle j=i+1; \rangle j \doteq i+1$$

is valid in $\mathcal{S}_{KeY}$ and provable in our calculus. As explained above this is reasonable as the premiss $i > \text{MAX\_T}$ is never true during the actual execution of a Java program.

In $\mathcal{S}_{KeY}$, the semantics of the built-in types `byte`, `short`, `int`, `long` and the operators acting on them exactly corresponds to semantics $\mathcal{S}_{Java}$ and thus to the

Table 2. Validity of sample formulas in the different semantics.

	$S_{OCL}, \mathbb{Z}$	S_{Java}	S_{KeY}	
			arithInt	int
Effects of overflow				
$\langle i = \text{MAX\_int} + 1; \rangle i \doteq \text{MIN\_int}$		✓		✓
$\langle i = 0; \text{ while } (i > 0) \ i++; \rangle \text{true}$		✓		✓
$\exists i. (i > 0 \wedge \langle i = i + 1; \rangle i < 0)$		✓		✓
$\forall i. \langle i = i + 1; \rangle i \doteq i + 1$	✓			
Incidentally correct programs				
$i > 0 \rightarrow \langle i = i + 1; \ i = i - 1; \rangle i > 0$	✓	✓		✓
$\forall i. \text{even}(i) \rightarrow \langle i = i + 2; \rangle \text{even}(i)$	✓	✓		✓
Effects of unreal states				
$\forall i. \langle i = i; \rangle i \doteq i$	✓	✓	✓	✓
$\forall i. (\langle j = i + 1; \rangle j \doteq i + 1)$	✓			
$\forall i. (i > \text{MAX\_T} \rightarrow \langle j = i + 1; \rangle j \doteq i + 1)$	✓	✓	✓	✓
$\forall i. (i > \text{MAX\_T} \rightarrow \text{false})$		✓		✓
Pure first-order formulas				
$\forall i. i + 1 > i$	✓		✓	
$\forall i. \exists j. j > i$	✓		✓	

definitions in the JAVA language specification. Hence, using the built-in types, it is still possible to make use of the effects of overflow by explicitly using the primitive built-in JAVA types in both the specification and the implementation.

In Table 1, properties of the combined semantics $\mathcal{S}_{KeY}$ are compared to those of $\mathcal{S}_{OCL}$ and $\mathcal{S}_{Java}$.

Table 2 shows in which of the different semantics some sample formulas are valid. For $\mathcal{S}_{KeY}$, the cases that the program variables i , j are of type `arithInt` resp. `int` are distinguished.

3.3 Properties of the Combined Semantics

In real states, the semantics $\mathcal{S}_{Java}$ of the built-in types corresponds to the semantics $\mathcal{S}_{Java'}$ of the arithmetical types. Thus, a program p , whose initial state is a real state, is equivalent to a program $p_{\text{transf}}(p)$, where the arithmetical types are replaced with the corresponding built-in types (see Theorem 3).

Corollary 1 summarises the important properties of $\mathcal{S}_{KeY}$. It states that, if the formula $\Gamma \rightarrow \langle p \rangle \psi$ is valid in $\mathcal{S}_{KeY}$ and the program p is started in a real state s such that $s \models_{\mathcal{S}_{KeY}} \Gamma$, no overflow occurs during the execution of the transformed program $p_{\text{transf}}(p)$ on the JAVA virtual machine, and after the execution the property ψ holds.

Note, that Corollary 1 does not apply to arbitrary formulas. For example, a formula of the form $[p] \text{true}$ is always derivable, whether overflow occurs during the execution of p or not. However, the generation of proof obligations for the correctness of a method typically results in formulas of the form $\Gamma \rightarrow \langle p \rangle \psi$ (Γ results from the pre-condition, ψ from the post-condition, and p is the implementation), so this is not a real restriction in practice.

The following theorems show that the differences between the verified program p and the actually executed program $p_{\text{transf}}(p)$ do not affect the verified behaviour of p .

Theorem 2. *If $\models_{S_{KeY}} \phi$, then both $\models_{S_{OCL}} \phi$ and $\models_{S_{Java'}} \phi$.*

Definition 4. *Let s be a real JavaDL state. The isomorphic state $iso(s)$ to s is the JVM state in which all state elements (program variables and fields) with an arithmetical type in s are of the corresponding built-in type and are assigned the same values as in s .*

If s is a real state, the existence of $iso(s)$ is guaranteed, since by definition, in real states the values of all variables of the arithmetical types are representable in the corresponding built-in types. In the following theorem, $s \llbracket p \rrbracket_{S_{Java'}}, s'$ means that program p , started in state s , terminates in state s' using semantics $S_{Java'}$.

Theorem 3. *Let p be a JAVA program that may contain arithmetical types. Then, for all real states s and all (arbitrary) states s' : If $s \llbracket p \rrbracket_{S_{Java'}}, s'$, then $iso(s) \llbracket p_{\text{transf}}(p) \rrbracket_{S_{Java}} iso(s')$.*

Corollary 1. *Let Γ, ψ be pure first-order predicate logic formulas, let p be an arbitrary JAVA program that may contain arithmetical types, and let s be an arbitrary JavaDL state.*

If (i) $\models_{S_{KeY}} \Gamma \rightarrow \langle p \rangle \psi$, (ii) $s \models_{S_{KeY}} \Gamma$, and (iii) s is a real state, then, when the transformed program $p_{\text{transf}}(p)$ is started in $iso(s)$ on the JVM, (a) no overflow occurs and (b) the execution terminates in a state in which ψ holds.

3.4 Variants of the Combined Semantics

In the definition of semantics S_{KeY} , the method `overflow` remains unspecified. By giving a partial specification, i.e., axioms that `overflow` must satisfy, it is possible to define variants of S_{KeY} . That way, one can allow certain occurrences of overflow, namely those which can be shown to be “harmless” using the additional axioms.

For example, one can define that the method `overflow` always terminates or implements an operation that is symmetric w.r.t. its arguments. If an axiom is added that `overflow` always terminates, a formula like $\Gamma \rightarrow \langle p \rangle \text{true}$ can be valid, even if overflow occurs during the execution of p , since goals of the form $\Gamma \vdash \langle x = \text{overflow}(\text{arg1}, \text{arg2}, \text{op}); \rangle \text{true}$ can immediately be closed using the information that the invocation of `overflow` terminates. That is, using such an axiom all overflow occurrences are defined to be “harmless” in cases where we are only interested in termination.

As long as the additional axioms are satisfiable by the instances S_{OCL} and $S_{Java'}$ of S_{KeY} , Theorem 2 and Theorem 3 still hold.

3.5 Steps in Software Development

Following our approach, the steps in software development are the following.

1. *Specification:* In the UML/OCL specification, the OCL type INTEGER is used.
2. *Implementation:* If an operation is specified using INTEGER, in the implementation, the arithmetical types `arithByte`, `arithShort`, `arithInt`, or `arithLong` are used.
3. *Verification:* Using our sequent calculus (see Section 4), one has to derive the proof obligations generated from the specification and implementation using the translation described in [5]. If all proof obligations are derivable, then Corollary 1 implies that the program, if the requirements of the corollary are satisfied and after replacing the arithmetical types with the corresponding built-in types, satisfies all specified properties during the execution on the JVM and in particular, no overflow occurs.

4 Sequent Calculus for the Combined Semantics

4.1 Overview

As already explained, the KeY system's deduction component uses the program logic JavaDL, which is a version of Dynamic Logic modified to handle JAVA CARD programs [4]. We have extended and adapted that calculus to implement our approach to handling integer arithmetic using the semantics $\mathcal{S}_{KeY}$.

Here, we cannot list all rules of the adapted calculus (they can be found in [21]). To illustrate how the calculus works, we present some typical rules representing the two different rule types: program transformation rules to evaluate compound JAVA expressions (Section 4.4) and rules to symbolically execute simple JAVA expressions (Section 4.5).

The semantics of the rules is that, if the premisses (the sequent(s) at the top) are valid, then the conclusion (the sequent at the bottom) is valid. In practice, rules are applied from bottom to top: from the old proof obligation, new proof obligations are derived.

Sequents are notated following the scheme

$$\phi_1, \dots, \phi_m \vdash \psi_1, \dots, \psi_n ,$$

which has the same semantics as the formula

$$(\forall x_1) \dots (\forall x_k) ((\phi_1 \wedge \dots \wedge \phi_m) \rightarrow (\psi_1 \vee \dots \vee \psi_n)) ,$$

where $x_1, \dots, x_k$ are the free variables of the sequent.

4.2 Notation for Rule Schemata

In the following rule schemata, *var* is a local program variable (of an arithmetical type) whose access cannot cause side-effects. For expressions that potentially have side-effects (like, e.g., an attribute access that might cause a `NullPointerException`) the rules cannot be applied and other rules that evaluate the complex expression and assign the result to a new local variable have to be applied first. Similarly, *simp* satisfies the restrictions on *var* as well or it is an integer literal (whose evaluation is also without side-effects). There is no restriction on *expr*, which is an arbitrary JAVA expression of a primitive integer type (its evaluation may have side-effects).

The predicate $in_T(x)$ expresses that x is in valid range, i.e.,

$$in_T(x) \equiv MIN\_T \leq x \wedge x \leq MAX\_T .$$

The rules of our calculus operate on the first *active* statement p of a program $\pi p \omega$. The non-active prefix π consists of an arbitrary sequence of opening braces “{”, labels, beginnings “try{” of try-catch-finally blocks, and beginnings “method-frame(...){” of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active statement is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately. The postfix ω denotes the “rest” of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the JAVA block “`1:{try{ i=0; j=0; }finally{ k=0; }}`”, operating on its first active statement “`i=0;` ”, then the non-active prefix π is “`1:{try{`” and the “rest” ω is “`j=0; }finally{ k=0; }}`”. Prefix, active statement, and postfix are automatically highlighted in the KeY prover as shown in Figure 1.

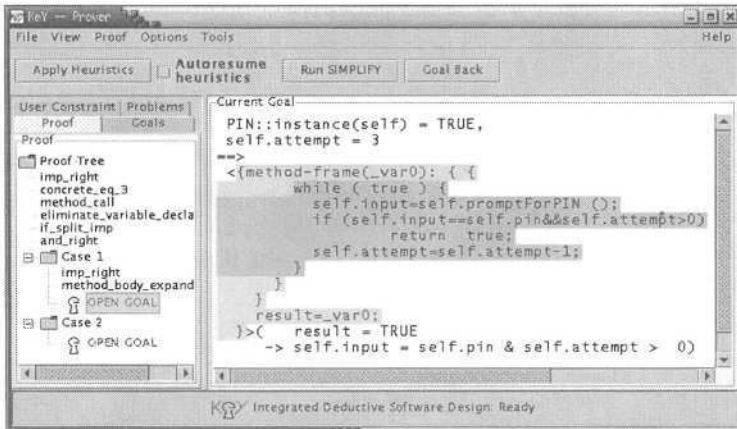


Fig. 1. KeY prover window with the proof obligation generated from the example.

4.3 State Updates

We allow *updates* of the form $\{x := t\}$ resp. $\{o.a := t\}$ to be attached to terms and formulas, where x is a program variable, o is a term denoting an object with attribute a , and t is a term (which cannot have side-effects). The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\{x := t\}\phi$ has the same semantics as $\langle x = t; \rangle\phi$.

4.4 Program Transformation Rules

The Rule for Postfix Increment. This rule transforms a postfix increment into a normal JAVA addition.

$$\frac{\Gamma \vdash \langle \pi \text{ var} = (T)(\text{var} + 1); \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \text{ var}++; \omega \rangle \phi, \Delta} \quad (\text{R1})$$

T is the (declared) type of var . The explicit type cast is necessary since the arguments of $+$ are *promoted* to `int` or `long` but the postfix increment operator $++$ does not involve promotion.

The Rule for Compound Assignment. This rule transforms a statement containing the compound assignment operator $+=$ into a semantically equivalent statement with the simple assignment operator $=$ (again, T is the declared type of var).

$$\frac{\Gamma \vdash \langle \pi \text{ var} = (T)(\text{var} + \text{expr}); \omega \rangle \phi, \Delta}{\Gamma \vdash \langle \pi \text{ var} += \text{expr}; \omega \rangle \phi, \Delta} \quad (\text{R2})$$

For the soundness of both (R1) and (R2), it is essential that var does not have side-effects because var is evaluated twice in the premisses and only once in the conclusions.

4.5 Symbolic Execution Rules

For the soundness of the following three rules it is important that var and simp are of an arithmetical type (rules for the built-in types can be found in [21]).

The Rule for Type Cast to an Arithmetical Type. A type cast from the declared type S of simp to an arithmetical type T causes overflow if the value of simp is in valid range of S but not in valid range of T (second premiss).

$$\frac{\begin{array}{l} \Gamma, \text{in}_S(\text{simp}) \rightarrow \text{in}_T(\text{simp}) \vdash \{ \text{var} := \text{simp} \} \langle \pi \omega \rangle \phi, \Delta \\ \Gamma, \text{in}_S(\text{simp}), \neg \text{in}_T(\text{simp}) \vdash \\ \quad \langle \pi \text{ var} = \text{overflow}(\text{simp}, \text{"cast}(T)"); \omega \rangle \phi, \Delta \end{array}}{\Gamma \vdash \langle \pi \text{ var} = (T)\text{simp}; \omega \rangle \phi, \Delta} \quad (\text{R3})$$

The Rule for Unary Minus. The unary minus operator only causes overflow if the value of var is equal to $\text{MIN\_}T$ (where T is the promoted type of var).

$$\frac{\begin{array}{l} \Gamma, \neg \text{simp} \doteq \text{MIN\_}T \vdash \{var := -\text{simp}\} \langle \pi \omega \rangle \phi, \Delta \\ \Gamma, \text{simp} \doteq \text{MIN\_}T \vdash \langle \pi var = \text{overflow}(\text{simp}, "-"); \omega \rangle \phi, \Delta \end{array}}{\Gamma \vdash \langle \pi var = -\text{simp}; \omega \rangle \phi, \Delta} \quad (\text{R4})$$

The Rule for Subtraction. This rule symbolically executes a subtraction and checks for possible overflow.

$$\frac{\begin{array}{l} \Gamma, \text{in}_{T_1}(\text{simp}_1) \wedge \text{in}_{T_2}(\text{simp}_2) \rightarrow \text{in}_T(\text{simp}_1 - \text{simp}_2) \vdash \\ \quad \{var := \text{simp}_1 - \text{simp}_2\} \langle \pi \omega \rangle \phi, \Delta \\ \Gamma, \text{in}_{T_1}(\text{simp}_1), \text{in}_{T_2}(\text{simp}_2), \neg \text{in}_T(\text{simp}_1 - \text{simp}_2) \vdash \\ \quad \langle \pi var = \text{overflow}(\text{simp}_1, \text{simp}_2, "-"); \omega \rangle \phi, \Delta \end{array}}{\Gamma \vdash \langle \pi var = \text{simp}_1 - \text{simp}_2; \omega \rangle \phi, \Delta} \quad (\text{R5})$$

The first premiss applies in case (1) both arguments and the result are in valid range (no overflow can occur) and (2) one of the two arguments is not in valid range (overflow is “allowed” as the initial state is already an unreal state).

The second premiss states that, if the arguments are in valid range but the result is not, the result of the arithmetical JAVA operation is calculated by the unspecified method `overflow`.

5 Extended Example

In this example we describe the specification, implementation, and verification of a PIN-check module for an automated teller machine (ATM). Before we give an informal specification, we describe the scenario of a customer trying to withdraw money.

After inserting the credit card, the user is prompted for his PIN. If the correct PIN is entered, the customer may withdraw money and then gets the credit card back. Otherwise, if the PIN is incorrect, two more attempts are left to enter the correct PIN. When an incorrect PIN has been entered more than two times, it is still possible to enter more PINs but even if one of these PINs is correct, no money can be withdrawn and the credit card is retained to prevent misuse.

Our PIN-check module contains a boolean method `pinCheck` that checks whether the PIN entered is correct and the number of attempts left is greater than zero. The informal specification of this method is, that the result value is `true` only if the PIN entered is correct and the number of attempts left is positive (it is decreased after unsuccessful attempts).

The formal specification of the method `pinCheck` consists of the OCL pre-/post-conditions

```

context PIN::pinCheck(input:Integer):Boolean
pre:  attempt=3
post: result=true implies input=pin and attempt>0

```

stating, under the assumption that `attempt` is equal to three in the pre-state, that `input` (the PIN entered) is equal to `pin` (the correct PIN of the customer) and the number of attempts left is greater than zero if the return value of `pinCheck` is true.

In this example the above formal specification is not complete (with respect to the informal specification):⁷ The relation between the attribute `attempt` and the actual number of attempts made to enter the PIN (invocations of the method `promptForPIN`) is not specified. The implicit assumption is that the number of attempts made equals `3 - attempt`. As we will see however, this assumption does not hold any more when decreasing `attempt` causes (unintended) overflow—leading to undesired results.

Without our additional arithmetical types, a possible implementation of the method `pinCheck` is the one shown on the left in Figure 2. Such an implementation may be written by a programmer who does not take overflow into account. This implementation of `pinCheck` basically consists of a non-terminating while-loop which can only be left with the statement “`return true;`”. In the body of the loop the method `promptForPin` is invoked. It returns the PIN entered by the user, which is then assigned to the variable `input`. In case the entered PIN is equal to the user’s correct PIN and the number of attempts left is greater than zero, the loop and thus the method terminates with “`return true;`”. Otherwise, the variable `attempt`, counting the attempts left, is decreased by one.

The generation of proof obligations from the formal OCL specification and the implementation yields the following JavaDL formula, where the body of method `pinCheck` is abbreviated with p :

$$\text{attempt} \doteq 3 \vdash \langle p \rangle (\text{result} \doteq \text{true} \rightarrow \text{input} \doteq \text{pin} \wedge \text{attempt} > 0)$$

Figure 1 shows this sequent after “unpacking” the method body of `pinCheck` in the KeY prover. This sequent is derivable in our calculus. Therefore, due to the correctness of the rules, it is valid in $\mathcal{S}_{KeY}$ and, thus, in particular in $\mathcal{S}_{Java}$. Consequently, the implementation can be said to be correct in the sense that it satisfies the specification.

But this implementation has an unintended behaviour. Suppose the credit card has been stolen and the thief wants to withdraw money but does not know the correct PIN. Thus, he has to try all possible PINs. According to the informal specification, after three wrong attempts any further attempt should not be successful any more. But if the thief does not give up, at some point the counter `attempt` will overflow and get the positive value `MAX_int`. Then, the thief has many attempts to enter the correct PIN and thus, to withdraw money.

⁷ In this simple example, the incompleteness of the specification may easily be uncovered but in more complex cases it is not trivial to check that the formal specification really corresponds to the informal specification.

<pre> class PIN { private int pin=1234; private int attempt; int input; public boolean pinCheck() { while (true) { input=promptForPIN(); if (input==pin && attempt>0) return true; attempt=attempt-1; } } } </pre>	<pre> class PIN { private int pin=1234; private arithInt attempt; int input; public boolean pinCheck() { while (true){ input=promptForPIN(); if (input==pin && attempt>0) return true; if (attempt>0) attempt=attempt-1; } } } </pre>

Fig. 2. Implementation of method `pinCheck` without (left) and with (right) using the additional arithmetical type `arithInt`.

The main reasons for this unexpected behaviour are the incomplete formal specification and the implementation that is “incidentally” correct w.r.t. the formal specification. In the following, we demonstrate that this unintended behaviour of the program can be detected following our approach.

In the implementation, we now use the arithmetical type `arithInt` for the variable `attempt` instead of the built-in type `int`. This results in a proof obligation similar to the one above. The only difference is that the variable `attempt` in the body of the method is now of type `arithInt` instead of `int`.

We do not show single proof steps and the corresponding rules that have to be applied. However, the crucial point in the proof is when it comes to handle the statement “`attempt=attempt-1;`”. After applying rule (R5) one of the new goals is the following:

$$in_{int}(attempt), in_{int}(1), \neg in_{int}(attempt - 1) \vdash \langle attempt = overflow(attempt, 1, "-"); \rangle \phi.$$

Since nothing is known about `overflow`, the only way to derive this in our JavaDL calculus is to prove—as a lemma or sub-goal—that no overflow occurs (and, thus, `overflow` is not invoked). Thus, one has to derive

$$in_{int}(attempt), in_{int}(1), \neg in_{int}(attempt - 1) \vdash false$$

or equivalently

$$in_{int}(attempt), in_{int}(1) \vdash in_{int}(attempt - 1) .$$

But the above sequent is neither valid nor derivable, because it is not true in states where `attempt` has the value `MIN_int`. In such states the subtraction

causes overflow and the sequent does not hold because its left side is true but its right side is false (as `attempt - 1` is not in valid range). The left part of Figure 3 shows the invalid sequent in the KeY prover.

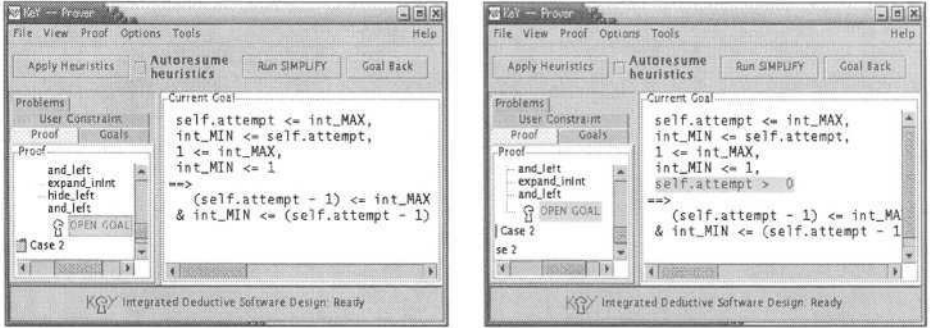


Fig. 3. The KeY prover window on the left shows an invalid sequent. The prover window on the right shows the same sequent with the additional highlighted premiss that makes the sequent valid.

Note, that this error is uncovered by using our additional arithmetical types and our semantics $\mathcal{S}_{\text{KeY}}$. If the built-in type `int` is used in the implementation, the error is not detected.

Since the proof obligation is not derivable in our calculus, one has to correct the implementation to be able to prove its correctness. For example, one can add a check whether the value of `attempt` is greater than 0 before it is decremented. This results in the implementation depicted on the right side in Figure 2. Trying to verify this new implementation with the KeY system leads to the sequent shown in the right part of Figure 3. In contrast to the one shown in the left part of Figure 3, this sequent is valid because of the additional formula `(self.attempt) > 0` on the left side, which stems from the added check in the revised implementation.

The resulting proof obligation can now be derived in our calculus and, thus, Corollary 1 implies that no overflow occurs if the type `arithInt` is replaced with `int` in order to execute the program on the JAVA virtual machine. With the improved implementation, it cannot happen that a customer has more than three attempts to enter the valid PIN and withdraw money since no overflow occurs.

To conclude, the main problem in this example is the inadequate (incomplete) specification, which is satisfied by the first implementation. Due to unintended overflow, this implementation has a behaviour not intended by the programmer. Following our approach, the unintended behaviour is uncovered and the program cannot be verified until this problem arising from overflow is solved.

As the example in this section shows, our approach can also contribute to detect errors in the specification. Thus, if a program containing arithmetical

types cannot be verified due to overflow, it should always be checked whether the specification is adequate (it may be based on implicit assumptions that should be made explicit).

6 Conclusion

We have presented a method for handling the data refinement relation between infinite and finite integer types. The main design goals of our approach are:

- “incidentally” correct programs are avoided by ensuring that no overflow occurs,
- the handling of the refinement relation is integrated into the verification calculus and, thus, hidden from the user as far as possible,
- the semantics combining both finite and infinite JAVA types provides a well-defined theoretical basis for our approach.

Acknowledgement. We thank R. Bubel, A. Roth, P. H. Schmitt, and the anonymous referees for important feedback on drafts of the paper.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. *Software and System Modeling*, pages 1–42, 2004. To appear.
3. W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, and P. H. Schmitt. The KeY approach: Integrating object oriented design and formal verification. In M. Ojeda-Aciego, I. P. de Guzman, G. Brewka, and L. M. Pereira, editors, *Proceedings, Logics in Artificial Intelligence (JELIA), Malaga, Spain*, LNCS 1919. Springer, 2000.
4. B. Beckert. A dynamic logic for the formal verification of Java Card programs. In I. Attali and T. Jensen, editors, *Java on Smart Cards: Programming and Security. Revised Papers, Java Card 2000, International Workshop, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
5. B. Beckert, U. Keller, and P. H. Schmitt. Translating the Object Constraint Language into first-order predicate logic. In *Proceedings, VERIFY, Workshop at Federated Logic Conferences (FLoC), Copenhagen, Denmark*, 2002. Available at: <http://www.key-project.org/doc/2002/BeckertKellerSchmitt02.ps.gz>.
6. N. Cataño and M. Huisman. Formal specification and static checking of Gemplus’ electronic purse using ESC/Java. In L.-H. Eriksson and P. A. Lindsay, editors, *Proceedings, FME 2002: Formal Methods - Getting IT Right, Copenhagen, Denmark*, LNCS 2391, pages 272–289. Springer, 2002.
7. P. Chalin. Improving JML: For a Safer and More Effective Language. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings, FME 2003: Formal Methods, Pisa, Italy*, LNCS 2805, pages 440–461. Springer, 2003.

8. ESC/Java (Extended Static Checking for Java).
<http://research.Compaq.com/SRC/esc/>.
9. European Space Agency. Ariane 501 inquiry board report, July 1996. Available at: <http://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf>.
10. S. Glesner. Java Card Integer Arithmetic: About an Inconsistency and Its Algebraic Reason, 2004. Draft.
11. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000.
12. J. Harrison. A machine-checked theory of floating point arithmetic. In Y. Bertot, G. Dowek, A. Hirschowitz, C. Paulin, and L. Théry, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs), Nice, France*, LNCS 1690, pages 113–130. Springer, 1999.
13. J. Harrison. Formal verification of IA-64 division algorithms. In M. Aagaard and J. Harrison, editors, *Proceedings, Theorem Proving in Higher Order Logics (TPHOLs)*, LNCS 1869, pages 234–251. Springer, 2000.
14. J. He, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, *European Symposium on Programming*, volume LNCS 213, pages 187–196. Springer, 1986.
15. M. Huisman. *Java Program Verification in Higher-Order Logic with PVS and Isabelle*. PhD thesis, University of Nijmegen, The Netherlands, 2001.
16. B. Jacobs. Java's Integral Types in PVS. In E. Najim, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of LNCS, pages 1–15. Springer, 2003.
17. A. B. Kit. Available at:
http://www.gemplus.com/smart/r_d/publications/case-study/.
18. G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
19. Object Management Group, Inc., Framingham/MA, USA, www.omg.org. *OMG Unified Modeling Language Specification, Version 1.3*, June 1999.
20. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential java. In S. D. Swierstra, editor, *Proceedings, European Symposium on Programming (ESOP), Amsterdam, The Netherlands*, LNCS 1576, 1999.
21. S. Schlager. Handling of Integer Arithmetic in the Verification of Java Programs. Master's thesis, Universität Karlsruhe, 2002. Available at: <http://www.key-project.org/doc/2002/DA-Schlager.ps.gz>.
22. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
23. K. Stenzel. Verification of JavaCard Programs. Technical report 2001-5, Institut für Informatik, Universität Augsburg, Germany, 2001. Available at: <http://www.informatik.uni-augsburg.de/swt/fmg/papers/>.
24. A. Stump, C. W. Barrett, and D. L. Dill. CVC: A Cooperating Validity Checker. In E. Brinksma and K. G. Larsen, editors, *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer-Verlag, 2002. Copenhagen, Denmark.
25. D. von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001.

Constituent Elements of a Correctness-Preserving UML Design Approach

Tiberiu Seceleanu and Juha Plosila

University of Turku, Department of Information Technology,
Laboratory of Electronics and Communication Systems
FIN-20014 Turku, Finland,
{Tiberiu.Seceleanu, Juha.Plosila}@utu.fi

Abstract. The correctness of design decisions is a very relevant aspect of building any software or hardware system. Emerging techniques tend to include formal methods in the system design flow. Together with older, established techniques, already well known to the present day designer, the combined approach should bring benefits in the form of correctness of the design, increase of reliability, etc, all these leading to a similar increase in productivity. In this study, we present a method of such combined design, by mixing a formal method strategies and rules, with UML, a relatively new but popular design method. Our formal framework is represented by the Action Systems formalism. We show how the UML models can be correctly changed by incorporating precise derivation rules expressed in OCL. The initial, abstract models can be thus transformed into more concrete models, without violating the intended specification.

Keywords: Action Systems, UML, refinement

1 Introduction

The complexity of modern day devices brings into the design flow problems related to correctness of design decisions, both in software as well as in hardware approaches. Formal methods proved to be useful when used in conjunction with older, established methodologies, and tend to be wider incorporated into the emerging techniques.

One of such techniques is represented by the Unified Modeling Language (UML) [16]. UML has become one of the most popular visual methods used in the development of a wide range of applications, which may be software, hardware, or mixed designs. The popularity comes from a rich set of language features and a very lax semantics, which actually allows every designer (group) to develop its own UML-based environment, and still be generally understood by a different group of designers. However, challenges arise when analyzing correctness aspects of any UML-based development flow.

UML itself contains resources for a more precise design environment. This is represented (since version 1.3) by the Object Constraints Language OCL [15],

a textual extension of UML. By employing a verbose syntax and simple logic constructs, OCL also targets developers who do not have a strong mathematical background. Initially, OCL was intended as a means to specify *invariants* of the systems under development and express *well-formedness* rules.

However, if one wants to develop incremental UML based system design, it would be very useful to be able to carry out refinement in UML and express it graphically, as well as to prove the correctness of each step. For this purpose, in this study we aim to improve on previous work that defines a UML profile for Action Systems [14]. Action Systems, introduced by Back and Kurki-Suonio [1] is a state-based formalism, relying on an extended version of Dijkstra's language of *guarded commands* [7]. Therefore, it behaves according to Dijkstra's guarded iteration statement on the state variables. The higher-order logic refinement calculus theory [3] can be used for reasoning about properties of action systems. An abstract specification of a program can be refined into concrete, implementable models, within this framework. The refinement calculus is also used for proving that specifications of abstract modules are preserved by their implementations.

A graphical notation is introduced by the profile, with interpretations given in terms of Action Systems. Each action composition has a unique graphical equivalent. A system is a collection of such action compositions. Our contribution stands in the introduction of the rules that interpret the refinement techniques within the mentioned profile, inducing changes in the graphical notation, too. Both at the action and at the system level, the designer may choose to make changes, by following certain rules. Our intention is to define the implication of these design rules on the UML model and, consequently, observed in the graphical notation. Hence, starting from concepts such as "action refinement" and proceeding towards system refinement we build a correctness preserving transformation methodology applicable to any design developed in the common Action Systems – UML framework.

When the term *refinement* is used in conjunction with UML, it means most usually a transformation of the UML model [12], possibly including a change of the behavior. *Refactorings* also represents a change in the model [13], with the requirement that behavior does not change. Our view on refinement considers, on the opposite, a *fixed* model. The targets are the instances specified by a certain design. We show how the possible changes in system behavior must be interpreted and processed, so that we preserve the consistency and correctness of the (graphical) system description.

As a result of this study, on one hand, we enhance the correctness reasoning capabilities of UML designs, and on the other hand, we expose the Action Systems formalism to a larger audience. Moreover, due to the numerous UML design tools, our work also establishes the basis for an automated design process, where correctness preserving transformations are validated by tools conforming to the rigorous framework of refinement calculus.

Overview of the paper. The rest of the paper is organized as follows. In section 2 we briefly introduce the reader into the Action Systems framework, by specifying a set of basic notions of the language. We proceed in section 3 with the description of the UML profile based on the Action Systems structure and semantics.

The main part of the study is represented by the content of section 4, presenting more details about the elements of the profile, refinement techniques and their impact on the UML representations. We continue with a design example in section 5, where we illustrate most of the aspects introduced in the previous sections. We end with some concluding remarks in section 6.

2 Action Systems

The *Action Systems* formalism, initially proposed by Back and Kurki-Suonio [1] and extended in other studies, is a framework for specification and correctness-preserving development of reactive systems. Based on an extended version of the guarded command language of Dijkstra [7], Action Systems is a state based formalism. It uses the *Refinement Calculus* [3] as the mathematical basis for disciplined stepwise derivation.

2.1 Actions

An *action* A is defined, for instance, by

$A ::= \text{skip}$	(empty statement)
$x := x'.R$	(non-deterministic assignment)
$x := e$	((multiple) assignment)
$A_1 \parallel \dots \parallel A_m$	(non-deterministic choice)
$A_1; \dots; A_m$	(sequential composition)
$A_1 // \dots // A_m$	(prioritized composition)
$P \rightarrow B$	(guarded action)

where $A_j, j = 1, \dots, m$, and B are actions, P and R are predicates (boolean conditions), x a (list of) variable(s) and e an (list of) expression(s).

Semantics of actions. An action A is considered *atomic*, that is, only its input-output behavior is of interest. This indicates that only the *initial* and *final* state of an action can be observed. Atomic actions may be represented by simple assignments or by more complex action compositions, such as the atomic sequence.

The *total correctness* of an action A with respect to a precondition P and a postcondition Q is denoted $P A Q$ and defined by

$$P A Q \triangleq P \Rightarrow \text{wp}(A, Q)$$

where $\text{wp}(A, Q)$ stands for the weakest *precondition* [7] of an action A to establish the postcondition Q .

We define for example:

$$\begin{aligned} \text{wp}(x := x'.R, Q) &\triangleq \forall x'. R \Rightarrow Q[x'/x] \\ \text{wp}(x := e, Q) &\triangleq Q[e/x] \\ \text{wp}(A_1 \parallel \dots \parallel A_m, Q) &\triangleq \text{wp}(A_1, Q) \wedge \dots \wedge \text{wp}(A_m, Q) \end{aligned}$$

The *guard* gA of an action A is defined by $gA \hat{=} \neg wp(A, false)$. An action A is said to be *enabled* in some state, if its guard is *true* in that state. Otherwise A is *disabled*. If gA is invariantly *true*, the action is *always enabled*. Further, the *body* of an action A is then computed as $sA \hat{=} gA \rightarrow A \parallel \neg gA \rightarrow abort$. Observe that sA is an always enabled statement. In our approach we will require *explicitly* such a characteristic for the body of any action. Therefore, in the case of a guarded action $A \hat{=} P \rightarrow B$ we consider $gA = P$ and $sA = B$. Observe next that any action A can be written in the form $true \rightarrow A$, and thus each action can be considered a guarded action, even though a non-trivial guard does not exist.

Non-atomic actions. Atomic action compositions do not always allow efficient modeling of complex system behavior. For example, when describing a communication sequence between systems, an atomic sequential composition cannot be used, as communication events take place in turns in separate atomic actions. In other words, the execution of a sequence is temporarily stopped in some system state, and resumed later in some other state. This kind of behavior is reflected by *non-atomic* action compositions, which allow the observation of intermediate states. In such a construct, the component actions may be atomic constructs of their own, but the composition in itself is not. A non-atomic composition is referred to as a *non-atomic action*.

2.2 Hierarchical Action Systems

An *action system* $\mathcal{A}$ is an iterative composition of actions executing on a set of local and global variables. A simplified hierarchical Action Systems model has the following form:

```

sys  $\mathcal{A}$  (interface list) [parameter list] ::
  || var local variable list
    invariant invariant list
    subsys subsystem instance list
    actions action list
    init initialization of variables
    exec do action composition od
      || subsystem composition
  ||

```

Execution of an action system. Starting with the original paper by Back and Kurki-Suonio [1], the sequential execution model was established as a *de facto* reasoning environment for AS designs. Parallel ($\parallel$) executions are modeled by interleaving actions that have no read / write conflicts.

An action system $\mathcal{A}$ operates as follows. The initialization places the systems in a stable, starting state, specified by the statements under the **init** clause. Enabled actions in the **do-od** loop or in the subsystem instances are subsequently selected for execution, one by one. Selection between simultaneously enabled

actions is nondeterministic. Parallel behavior is modeled by simultaneously enabled actions which can be *interleaved*, i.e., executed in any order. This is the case, if the actions in question do not write onto the same variables, and can neither disable nor enable each other. When all actions both in the **do-od** loop and in the subsystem instances become disabled, the computation stops temporarily until a system communicating with $\mathcal{A}$ enables one of the actions again via the interface variables.

2.3 Refinement Aspects

Action systems are meant to be designed in a stepwise manner within the *refinement calculus* framework [3]. The transformations of actions or systems performed under the *refinement calculus* rules preserve the correctness of the actions.

The (atomic) action A is said to be (*correctly*) *refined* by action C , denoted $A \leq C$, if $\forall q. (wp(A, q) \Rightarrow wp(C, q))$ holds. This is equivalent to the condition $\forall p, q. ((p \wedge A) \Rightarrow (p \wedge C \wedge q))$, which means that the *concrete* action C preserves every total correctness property of the *abstract* action A . In the following paragraphs we introduce certain basic notions used in the refinement procedures applied to actions.

Invariants. A predicate P is an *invariant* of an action A , if $P \Rightarrow wp(A, P)$. At the system level, P is an invariant of an action system $\mathcal{A}$ if it is established by the initialization and if it is an invariant of each of the component actions of $\mathcal{A}$.

Data Refinement. Assume two actions A and C with variables a, u and c, u , respectively. Let $R(a, c)$ be a boolean relation between the variables a and c , and $I(c, u)$ an invariant over the action C on the variables c and u . The abstract action A is *data-refined* by the concrete action C using the *abstraction relation* $R(a, c) \wedge I(c, u)$, denoted $A \leq_{R, I} C$, if

$$\forall q. (R \wedge I \wedge wp(A, q) \Rightarrow wp(C, \exists a. R \wedge I \wedge q)) \quad (1)$$

holds. The predicate $\exists a. R \wedge I \wedge q$ is a boolean condition on the program variables a and c .

We may reuse (1) even when the relation R is trivial, that is R is an identity relation of the possible kind $a = c$. This situation suggests then that there is no change in data representation, but some other, behavioral changes were envisaged, and is expressed by

$$\forall q. (I \wedge wp(A, q) \Rightarrow wp(C, \exists a. I \wedge q))$$

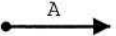
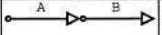
A change at the action level is always accompanied by a change in the system's data representation or behavior.

the system and manage large complex systems easier, without getting lost in the details. The operators connecting the actions and the action systems are clearly visible in the graphical representation and thus the overall functionality becomes apparent. The profile customizes and extends specific UML types for every action system operator and defines a notation for them. Each of the elements in Fig. 1 has an attribute, `isDrawn: Boolean`, which specifies if the element is drawable or not. For instance, we have that `ASDeclaration.isDrawn = false`, but `AS-ActionComposition.isDrawn = true`. The values of the `isDrawn` attribute are specified as invariants for the specific classes.

The graphical representation is intended to offer a fast and illustrative understanding of the system behavior. UML in itself provides similar methods, such as sequence, collaboration or activity diagrams. Employing such features, one could think of an even better integration of the Action Systems framework into the UML environment. However, mostly due to the interleaved execution model of Action Systems and the intrinsic non-determinism, such (deterministic) methods would not correctly represent the behavior of an Action Systems description. Therefore, we are compelled to build our own graphical interface and models.

In the following sections one should consider that there is a specified connection between the ASUML model of Fig. 1 and the graphical notations described in Table 1.

Table 1. Notation of actions

Name	Notation	AS Meaning
Atomic Action		A
Atomic Sequence		$A; B$
Non-Atomic Sequence		$A \ ; \ B$
Non-Deterministic Choice		$A \parallel B$
Prioritised Composition		$A // B$

4 The Design Elements of the ASUML Profile

In the following, we start by introducing the building blocks of the common design approach, Action Systems – UML. In sequence, we offer specifications on how the refinement techniques employed in Action Systems designs affect the modeling of these elements. For this, we use OCL-based methods. The features of this language offer us an elegant way to firstly impose well-formedness rules – expressed as *invariants* of classes or methods – and secondly, to describe the effects of transformations on all the necessary locations – by means of pre-

and post-conditions associated to the methods. As OCL, like UML, is under continuous development, some features are found in proposals for extension.

4.1 The ASAction Class

The model for the atomic actions of the Action Systems is represented by the ASAction class derived as presented in Fig. 1. An instance of the ASAction class constitutes the elementary building block of a design within the ASUML profile. Such an element of design does not contain any internal operator, that is, it is not composed of “smaller” elements of design. The class has several attributes and methods, presented in the following paragraphs.

Attributes of ASAction. The attributes are specified as:

- **Name:** String. This attribute represents the name associated with the specific ASAction instance.
- **System:** String. This attribute specifies the system that contains (has a link to) the specific ASAction instance.
- **readVar, writeVar:** Set (String). The variables read or updated by some action *A* are collected in these sets, respectively, together with their associated types.
- **Guard:** Boolean. The guard of the action is specified as a separate attribute, a boolean expression.
- **Body:** String. The updates to be performed by the actions are specified by this attribute.
- **Invariant:** Boolean. Certain actions may be required to respect specific, private invariants. These are also specified as attributes of the class, remaining to be also specified for each instance.
- **isDrawn:** Boolean. It indicates if the element may appear in a graphical representation, depending on the hierarchy level at which the action system is observed.

Invariants of ASAction. Every instance of the ASAction class must comply with certain requirements. For instance, it must have a name; it must be part of an action system, etc. We specify such constraints of the design by providing the ASAction class with an invariant which is supposed to be satisfied every time a new instance of ASAction is created. The invariant is illustrated as:

context ASAction inv Creation:

```

name <> '' and
System <> '' and
readVar = notEmpty and
writeVar = notEmpty and
Body <> '' and
if Guard = undefined then Guard = true endif and
isDrawn = true

```

Methods of ASAction. Here, we only analyze two methods of the ASAction class, used at different stages of the design, as presented below.

- ToAC(C: ASAction; n: String; o:operator): ASActionComposition. This method transforms an existent action into an action composition. Even though one would expect that all the transformations have a correctness supporting background (in the sense of refinement), especially in earlier phases of design, this may not be true. Hence, an operation such as the extension of an action into an action composition may often help us build an initial specification of the problem at hand, even though it may not represent a refinement step. However, the transformation of an ASAction into an ASActionComposition may usually be considered as a correctness preserving operation, for instance when introducing new variables and specifying their respective updates. The implications for the UML model following the execution of this method are illustrated as follows.

```

context ASAction:: ToAC(C: ASAction; n: String; o:operator): ASActionCom-
position
  let : InASD = ASDeclaration.ASAction-> select(a | a.name = name)
  let : InASAC = ASActionComposition.ASAction-> select(a | a.name =
name)
  let : InEL = ExecutionLoop.ASAction-> select(a | a.name = name)
  pre : name <> C.name and
  post : AC.oclIsNew() and
    if n = '' then AC.name = name else AC.name = n endif and
    InASD.name = AC.name and
    InASAC.name = AC.name and
    InEL.name = AC.name and
    ExecutionLoop.Var=(ExecutionLoop.Var@pre - readVar@pre - write-
      Var@pre)-> union(AC.readVar->union (AC.writeVar)) and
    AC.SubComp = { self, C } and
    AC.operator = o and
    AC.readVar = readVar->union(AC.readVar) and
    AC.System = self.System and
    AC.System -> AddA(C) and AC.System -> AddA(AC) and
    self.System -> RemA(self) and
    --- completing the composition attributes
  ...

```

The constraints imposed on the above method state that, before extending the action, we have to check that we do not duplicate the same action and that the system name also changes, hence, the new one must not be empty. After the execution of the method, a new action composition is created, with an appropriate, possibly the same name; the previous action becomes a subcomponent of the new composition, together with the added action; the operator of the composition comes as a parameter, etc. The *collections* InASD, InASAC, InEL are intended to group all the occurrences of the initial action within the containing system: in the declaration part, as a stand alone action, in any of the compositions, or in the execution loop contained by the body. At the system level, the new composition together with the new action *C* are inserted in the **actions** clause, while the initial action is removed.

• `Refine(C:ASAction, R:relation) : Boolean`. The method takes as parameters the concrete action C and the abstraction relation R . The body of the *Refine* method returns the validity of the relation $A \leq_R C$ and it is based on the refinement relation expressed by (1). The result is a *true* or *false* value, depending on the correctness of the transformation.

In the context of the ASUML profile, certain additional constraints on performing the refinement are expressed by:

```
context ASAction:: Refine(C: ASAction, R: Boolean) : Boolean
  let : InASD = ASDeclaration.ASAction-> select(a | a.name = name)
  let : InASAC = ASActionComposition.ASAction-> select(a | a.name =
name)
  let : InEL = ExecutionLoop.ASAction-> select(a | a.name = name)
  pre : name <> C.name
  post : self = C and
    InASD.name = C.name and
    InASAC.name = C.name and
    InEL.name = C.name and
    ExecutionLoop.Var=(ExecutionLoop.Var@pre - readVar@pre - write-
Var@pre) -> union(C.readVar->union (C.writeVar))
```

The skip action. A special kind of action is the one that intrinsically preserves the state of the system. This action is generically called *Askip*. The body of *Askip* contains only the command *skip*, and its *Guard* is *true*. The *readVar*, *writeVar* attributes contain all of the existing variables in the design.

4.2 The ASActionComposition Class

An action composition, represented by an instance of the *ASActionComposition* class, is a complex behavior that unifies several actions by means of a **single** operator.

Attributes of ASActionComposition. The attributes are specified as:

- *Name*, *System*, *ReadVar*, *WriteVar*, *Invariant*, *Guard*, *isDrawn*. The same as for *ASAction*.
- *SubComp*: `Collection(ASActionComposition)`. This set unites the component actions or action compositions of an instance of *ASActionComposition*. Depending on the type of the operator, the order in this collection may be important.
- *Operator*: `String`. This denotes the name of the operator used in building the composition.

The attributes of an action composition, except for *Name* and *System*, are obtained by parsing and collecting the correspondent attributes of the composing actions. For instance, the *Guard* and the *Body* of an action composition are computed based on the guards and bodies of the components. The computation also depends on the operator of the composition. In the following, we exemplify the computation of the *Guard* and the *readVar* attributes of an action composition. The construction of these two attributes is offered as an *invariant*:

```

context ASActionComposition inv Guard:
  if operator = '' then
    Guard = SubComp.Guard
  elsif operator = 'choice' then
    Guard = SubComp->iterate(e:ASActionComposition; g: Boolean = true
      | g or e.Guard)
  elsif operator = 'simultaneous' then
    Guard = SubComp->iterate(e:ASActionComposition; g: Boolean = true
      | g and e.Guard)
  -- other operators to follow...
  ...
endif

context ASActionComposition inv readVar:
  readVar = ASAction->iterate(e:ASAction; g: Set(String) = ''
    | g->union(e.readVar))

```

Notice that, if the operator attribute is set to '' then the SubComp collection must have a single element. Consequently, that instance of ASActionComposition can be evaluated as an ASAction:

```

context ASActionComposition inv Size:
  operator = '' implies SubComp->size = 1

```

Methods of ASActionComposition. Similar to the case of simple actions, we only analyze here the refinement method of the action composition and the extension of composition with another action (composition).

- **AddA**(C: ASActionComposition; o:operator;n: String). We may often need, especially in earlier phases of design to extend initial behaviors in order to cover the system requirements. While this may not be a traditional refinement step, the changes must be reflected in the UML model. Hence, the method AddA provides us with such features. Notice that in advanced stages of the design, such additions may appear as results of the trace refinement method described in the next section.

The characteristics of the AddA method are expressed as:

```

context ASActionComposition:: AddA(C: ASActionComposition; o:operator;
n: String) : ASActionComposition
  pre : name <> C.name
  post : Name = n and
    SubComp = {self.SubComp, C } and
    operator = o and
    ASDeclaration.Var->unite(C.readVar-> unite(C.writeVar))

```

- **Refine**(C: ASActionComposition;R: Boolean). Besides the actual refinement relation that has to be validated, expressed by relation (1), there are changes that have to be reflected in the UML model as well. Considering a generic refinement $A \leq_R C$, where both A and C are action compositions, these changes are specified as:

```

context ASActionComposition:: Refine(C: ASActionComposition; R: Boolean)
: Boolean

```

```

let : InASD = ASDeclaration.ASActionComposition->select(a|a.name =
name)
let : InASAC = ASActionComposition. ASActionComposition->select(a|
a.name = name)
let : InEL = ExecutionLoop.ASActionComposition->select(a|a.name =
name)
pre : name <> C.name
post : self = C and
      InASD.name = C.name and
      InASAC.name = C.name and
      InEL.name = C.name

```

The transformations performed by refinement also have an impact on the graphical representation of the refined actions. It is of great importance that such transformations are valid, as in a graphical environment correctness issues are more difficult to visualize. Therefore, the Refine method, for the Action Systems part, together with the associated pre- and post-conditions, for the UML modeling part, ensure the correctness of the transformation. An illustration of applying the method to an action composition is given in the following example. *Example.* Consider the generic action composition $AC \triangleq (A; C) \parallel (B; C)$. One can prove that $AC \leq AC_1$, where $AC_1 \triangleq (A \parallel B); C$, as the refinement relation (1) holds for any A, B, C . Considering the above specification, the action AC_1 will replace AC and all the occurrences of AC in the declaration part, other action compositions and in the execution loop of the system AC .System. The refinement process is pictured in Fig. 2.

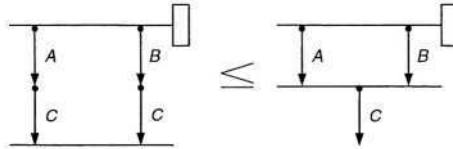


Fig. 2. Graphical refinement: $AC \leq AC_1$.

Notice how some of the characteristics of the action composition changed after the above transformation:

	before	after
operator :	$\parallel$	;
SubComp :	$\{A; C, B; C\}$	$\{A \parallel B, C\}$

Observe also that the example could have been specified the other way around, that is: $AC_1 \leq AC$.

4.3 The ASSystem Class

Basically, an action system is a collection of actions, operating on a set of global and local variables. The interface, the local declarations and the behavior are the

three main regions of an action system representation. In the profile, all three of them are introduced as distinct classes. Different operations on the system itself may affect only some of these classes, or all of them.

Next, we analyze several aspects of *trace refinement* of Action Systems as they are reflected in the ASUML profile. In a very succinct presentation, one may say that an action system $\mathcal{A}$ is trace refined by the action system $\mathcal{C}$ if some actions of $\mathcal{C}$ are the same as in $\mathcal{A}$, some other actions of $\mathcal{C}$ are refinements of corresponding actions of $\mathcal{A}$ and the system $\mathcal{C}$ may contain new (*auxiliary*) actions that refine skip in $\mathcal{A}$. There are more details concerning the trace refinement in Action Systems, but for these, the reader is directed to [2].

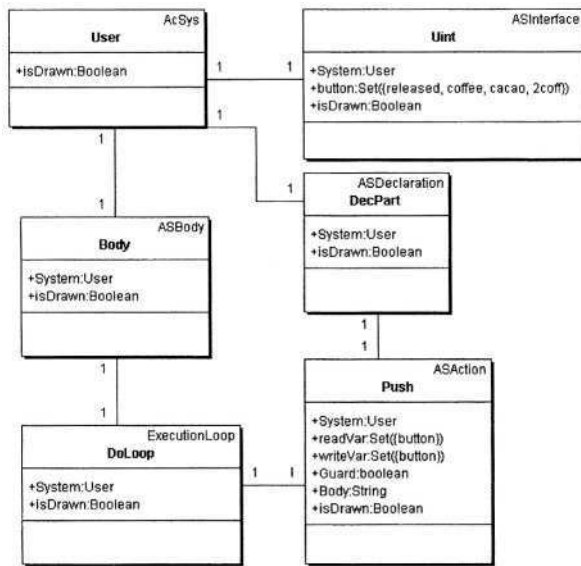


Fig. 3. The class diagram for the *User* system.

While, again, we do not focus on the body of an existing `TraceRef` method of a given system $\mathcal{A}$, we represent instead the implications of applying such a method on $\mathcal{A}$. Hence, the constraints imposed on the method are expressed as:

context ASSystem:: `TraceRef`(C: ASSystem; A_1 , X: ASActionComposition;

R: relation) : boolean

pre : (ExecutionLoop.ASAction->exists(a : ASAction| a.Refine(A_1 ; R)))

and ASkip.Refine(X)

post : X.ocllsNew() **and**

ASDeclaration->unite(X) **and**

ExecutionLoop.ASActionComposition = ExecutionLoop.ASActionComposition.AddA(X; choice; ExecutionLoop.ASActionComposition.Name)

tion.Name)

5 Design Example

In this section, we present a simple example that analyzes the design of a coffee machine serving either coffee (in single or double quantities) or cocoa. The concepts introduced before lay underneath the changes which occur in the graphical representation of the systems, during the design steps. While in the refinement related issues analyzed in the previous sections we did not specify any changes regarding the name of the systems in either action refinement or trace refinement, in the following, in order to keep track of the transformations at system level, we will change the names of the action systems after every refinement step.

We start by specifying the interface between a hypothetical user and the machine. The two action systems are described as follows:

```

sys User (button: {released, coffee, cocoa, 2coff}) ::
||
actions Push : button := b'.(b' ∈ {coffee, cocoa, 2coff})
init button := released
exec do Push od
||

sys Machine (button: {released, coffee, cocoa, 2coff}) ::
|| var counter: {0, 2}; source: {0, 1}; busy : boolean
actions
Sel0 : button = coffee → source = 0; counter := 1
Sel1 : button = cocoa → source = 1; counter := 1
Sel2 : button = 2coff → source = 0; counter := 2
BsyT : busy := true
Sel : (Sel0 || Sel1 || Sel2); BsyT
BsyF : busy := false
RelB : button := released
Delivery : busy → RelB; BsyF
init button := released; source, counter := 0; busy := false
exec do Sel || Delivery od
||

```

The corresponding class diagram for the system *User* is shown in Fig. 3, while the graphical notation for both the *User* and the *Machine* systems is shown in Fig. 4. A more detailed representation of the system *Machine* is shown in Fig. 5. The components of the selection action *Sel* are in their turn represented as action composition, with an immediate identification of the components (for instance, *Count*₁ : *counter* := 1, etc).

The system *Machine* is obviously more interesting to study than the *User*, as its functionality is more complex, and not yet fully described. As a next step, based on the analysis presented in the previous section, we can reshape the graphical representation, as illustrated in Fig. 6.

After the user pressed a button determining a selection, the machine should also start serving the requested beverage, before the resetting of the variable *button*. This is described by a new action composition, i.e., *Service*. In a very

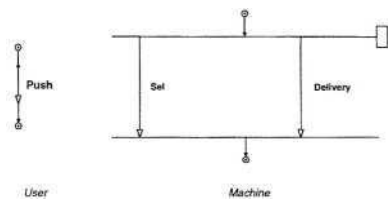


Fig. 4. The graphical notation of the two systems.

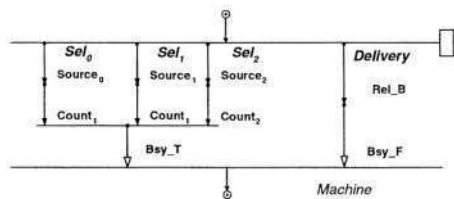


Fig. 5. A more detailed graphical representation of *Machine*.

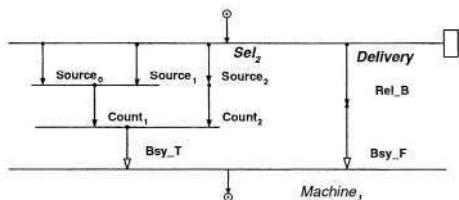


Fig. 6. Refined graphical notation of *Machine₁*.

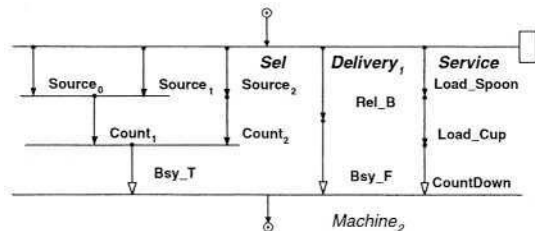


Fig. 7. The graphical notation of *Machine₂*.

intuitive manner, it fills a spoon, from the selected coffee or cocoa box (selection based on the value of the variable *source*) and then, the spoon is emptied into the cup. The operation is repeated for as many times as indicated by the value of *counter*. Notice that, when we introduce *Service*, also several new variables appear in the declaration part of the system.

The *Delivery* action also supports certain changes. More explicitly, the actual delivery should take place after the service completed, that is, in the case of a double amount of coffee, after the second spoon has been dropped into the cup. Hence, we replace *Delivery* with *Delivery₁*, by strengthening the guard of the initial action.

```

sys Machine2(button: {released, coffee, cocoa, 2coff}) ::
||
var counter, source_box[0..1]: integer;
    Spoon: { empty, full }; source: {0, 1}; busy : Boolean
const fullSpoon: integer := 10
actions
Source0      : button = coffee → source = 0
Source1      : button = cocoa → source = 1
Sel2        : button = 2coff → source = 0; counter := 2
Count1      : counter := 1
BsyT        : busy := true
Sel          : (((Source0 || Source1); Count1) || Sel2); BsyT
BsyF        : busy := false
RelB        : button := released
Delivery1    : busy ∧ counter = 0 → RelB; BsyF
CountDown    : counter > 0 → counter := counter - 1
Load_Spoon   : Spoon = empty ∧ source_box[source] > 0 →
               source_box[source] := source_box[source] - fullSpoon; Spoon := full
Load_Cup     : Spoon = full → Spoon := empty
Service      : counter > 0 → Load_Spoon; Load_Cup; CountDown
init counter := 0; button := released; Spoon := empty;
      source = 0; busy := false; source_box = 1000
exec do Sel || Delivery1 || Service od
||

```

The new graphical representation of system *Machine₂* is given in Fig. 7.

Another kind of transformation may still be applied, in this case concerning both the *Machine₂* and the *User* systems. Suppose that, instead of a button represented as in the context above, we imagine that actually there are three buttons allowing the user to place an order. Hence, the generic *button* variable can be replaced, in the next step, by a more concrete representation, as a vector with three elements, representing the three possible orders, each of them being allowed to have a value of *pressed* or *released*. To make this transformation, we use the abstraction relation *R*, that establishes the connection between the abstract variable *button* and the concrete one *buttons*[0 ... 2] : {*pressed*, *released*}, defined as follows:

$$\begin{aligned}
R \hat{=} & \text{button} = \text{coffee} \Rightarrow \text{buttons}[0] = \text{pressed} \\
& \wedge \text{button} = \text{cocoa} \Rightarrow \text{buttons}[1] = \text{pressed} \\
& \wedge \text{button} = \text{2coff} \Rightarrow \text{buttons}[2] = \text{pressed} \\
& \wedge \text{button} = \text{released} \Rightarrow \forall j \in \{0 \dots 2\}. \text{buttons}[j] = \text{released}
\end{aligned}$$

This change affects the actions *Push* and *Sel* (with its components). They are thus refined into *Push*₁ and *Sel*₁, described as follows:

$$\begin{aligned}
\text{Push}_1 : & \quad \text{buttons}[0] := b'.(b' = \text{pressed}) \\
& \parallel \text{buttons}[1] := b'.(b' = \text{pressed}) \\
& \parallel \text{buttons}[2] := b'.(b' = \text{pressed})
\end{aligned}$$

$$\begin{aligned}
\text{Source}_0^1 : & \quad \text{buttons}[0] = \text{pressed} \rightarrow \text{source} = 0 \\
\text{Source}_1^1 : & \quad \text{buttons}[1] = \text{pressed} \rightarrow \text{source} = 1 \\
\text{Sel}_2^1 : & \quad \text{buttons}[2] = \text{pressed} \rightarrow \text{source} = 0; \text{counter} := 2 \\
\text{Sel}_1 : & \quad (((\text{Source}_0^1 \parallel \text{Source}_1^1); \text{Count}_1) \parallel \text{Sel}_2^1); \text{Bsy}_T \\
\text{Rel}_B^1 : & \quad \text{buttons} := \text{released}
\end{aligned}$$

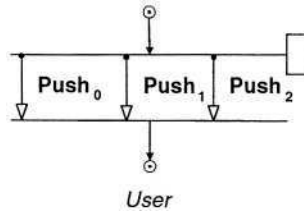


Fig. 8. The system *User*₁.

Graphically, only changes in the *User* system are visible (Fig. 8), while, up to the renaming of the actions, *Machine*₃ does not differ from the *Machine*₂ representation, and therefore we do not show it again. The equivalent changes in the class diagram of system *User*₁ can be observed in Fig.9.

6 Conclusions

In this study, we presented means by which the refinement techniques used for the Action Systems are applied within a common design framework shared with UML. The connection is obtained by employing OCL specified constraints, mostly by imposing pre- and post conditions on refinement related methods. These conditions, plus, whenever applicable and useful, invariants, ensure that a correct transformation performed in Action Systems has a correct representation in the graphical environment of the considered ASUML profile.

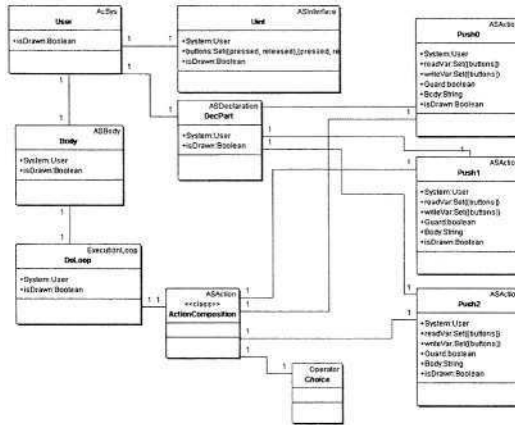


Fig. 9. The class diagram of system $User_1$.

We used the term *refinement* as we understand it in the Action Systems framework, that is, related to successive system transformations, and not in the usual manner in which UML designers may use it. Also different from the refactoring concept, we used a stable, *fixed* UML model, as described in Fig. 1. Based on this model, we construct our system instances, as required by a specific design target. Instead of working on classes, associations, use cases, etc., we concentrated on the use of such elements and the impact on changing behavioral aspects on the system description and not on the underlying model. Even though the example introduced here is not a very complex one, basic features of our refinement techniques and their immediate representation in the UML profile are illustrative. A more complex and important design situation, the analysis of a segmented bus arbitration unit, is presented in [10].

Related work. The search for graphical representation of otherwise difficult to follow (formal) languages is not a new subject. Various textual language paradigms have been given a visual form, recently targeting translations into UML descriptions.

One of the closest studies we could mention here is the one initiated by Bolton and Davies in [4], where UML activity diagrams are given both behavioral semantics in terms of CSP processes, and relational semantics as Z notations. By using the refinement ordering the authors can reason about the expected design behavior, information presented in possible different UML notations. In contrast, our approach ensures the consistency of a single design representation, at different levels of abstraction. Thus, we consider that our present analysis covers an immediate lower level compared to the cited work. One of the disadvantages is that we have to build our own tools in order to support the ideas introduced here. However, the progress of such approaches gives us hints on further development of our own design environment.

In [8], Hammad et al. interpret B machines as UML state diagrams. As in our case, the study is motivated by the same need for visual representation and

availability to the non-specialist. The refinement techniques come from the same family as the ones we discussed in the previous sections, that is, they refer to the systems under development and not to the UML model. However, in addition, our profile also describes the interaction between systems as illustrated in the design example, and the transformations are specified in OCL. The same formalism, B, and its connection to UML are analyzed by Sekerinski and Zurob in [11], where the UML state diagrams are emphasized, too. The parallel composition is different from our interleaved perspective, the result of such a construct being visible only in the next step.

By developing a new graphical environment, we come close to the solution adopted by Brooke and Paige in [5], where a newly defined visual representation is connected to the timed CSP formal framework. Still, as our approach is more integrated, that is, with UML models, we envisage a faster tool development process.

For a language in evolution, such as OCL, it is primordial to establish stable syntax and semantics. In a general overview of OCL features, Richters and Gogola [9] insist on the syntax and semantics of the pre- and post-conditions from a low logical level towards class associations and states of the system. Another view on the logical correctness of the basic constructs of OCL can be found in [6], where Brucker and Wolff analyze the embedding of OCL into a higher order logic framework. Compared to these studies, ours is more restricted, as it deals already with a high abstraction level, the Action Systems environment. The work presented here is more similar to the analysis developed for instance in [17], where aspects of queries, views and transformations are illustrated. Still, the cited paper contains a proposal for further extending UML environments, while we only tried to use such proposals. However, as the main lines are drawn already, we consider that forthcoming changes in both the UML and OCL frameworks can easily be adapted to the presented views.

Future work. The further development of the ASUML profile considers the actual realization of the association between the class model and the graphical representation, both illustrated in this study from a theoretical point of view. Based on this connection, tool support for the changes mentioned in the previous sections becomes a realizable task. Thus, changes on system representation would be available either as textual or graphical actions, with immediate effect on the complementary view, too. Given the precise rules that govern the system design in Action Systems, we believe that the transformations could be thus easily automated. However, there is an important aspect which, in our opinion cannot be overlooked at this point, and it is also observable in the present work. It consists of the computation of the result returned by the refinement methods. This, at least in our representation, requires something more than OCL and other UML related features in order to cover the background suggested by the refinement rules. Nevertheless, one may search for appropriate tools that can solve this aspect and configure their usage such that it could fit the expressed necessity. Naturally, this is one of the subjects targeted by further work in this direction.

Another important aspect of the forthcoming studies is the connection to the other methods of UML design environment, such as state, sequence and activity diagrams. For reasons based on the execution model of an action system, this is not an easy task and it was avoided, for the time being. However, another plus of employing UML is that it provides mechanisms through which the translation between the different possible views of a representation becomes an easier task.

Acknowledgements. The authors are thankful to the assigned reviewers, who helped us improve the content of the paper.

References

1. R. J. R. Back and R. Kurki-Suonio. *Distributed Cooperation with Action Systems*. In ACM Transactions on Programming Languages and Systems, Vol. 10, No. 4, 1988, pp. 513-554.
2. R. J. R. Back and J. von Wright. *Trace refinement of action systems*. CONCUR-94, Springer-Verlag, August 1994.
3. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
4. C. Bolton, J. Davies. *Using Relational and Behavioral Semantics in the Verification of Object Models*. In C. Talcott and S. Smith (Eds.), Proceedings of FMOODS 2000. Kluwer, 2000.
5. P. J. Brooke, R. F. Paige. *The Design of a Tool-Supported Graphical Notation for Timed CSP*. IFM 2002, LNCS Volume 2335 / 2002, pp. 299-318.
6. A. D. Brucker and B. Wolff. *HOL-OCL: Experiences, Consequences and Design Choices*. In J.-M. Jezequel and H. Hussmann and S. Cook (Eds.): UML 2002, LNCS, pp. 1-15, 2002. Springer-Verlag Berlin Heidelberg 2002.
7. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
8. A. Hammad, B. Tatibouët, J.-C. Voisinnet, W. Wu. *From a B Specification to UML StateChart Diagrams*. ICFEM 2002, LNCS Volume 2495 / 2002, pp. 511-522.
9. M. Richters and M. Gogola. *OCL: Syntax, Semantics and Tools*. In A. Clark and J. Warmer (Eds.): Object Modeling with the OCL, LNCS 2263, pp. 42-68, 2002. Springer-Verlag Berlin Heidelberg 2002.
10. T. Seceleanu, T. Westerlund. *Aspects of Formal and Graphical Design of a Bus System*. To appear in Proceedings of the Design Automation and Test in Europe Conference, 2004.
11. E. Sekerinski, R. Zurob. *Translating Statecharts to B*. IFM 2002, LNCS Volume 2335 / 2002, pp. 128-144.
12. D.F. D'Souza, A.C. Wills. *Objects, Components and Frameworks with UML -The Catalysis Approach*. Addison-Wesley Longman, 1999.
13. G. Sunyé, D. Pollet, Y. Le Traon, J.-M. Jézéquel. *Refactoring UML Models*. UML 2001, volume LNCS 2185, pp. 134-148. IRISA, Springer Verlag, 2001.
14. T. Westerlund, T. Seceleanu. *UML Profile for Action Systems*. TUCS technical report Nr. 581, 2003.
15. Object Management Group. *Object Constraint Language Specification*. Version 1.3, 1999.
16. Object Management Group. *Unified Modeling Language Specification*.
17. DSTC, IBM. *MOF Query / Views / Transformations*. Initial submission, March, 2003.

Relating Data Independent Trace Checks in CSP with UNITY Reachability under a Normality Assumption

Xu Wang¹, A.W. Roscoe¹, and R.S. Lazic²

¹ Oxford University Computing Laboratory,
Oxford OX1 3QD, UK. {xu.wang,bill.roscoe}@comlab.ox.ac.uk

² Department of Computer Science, University of Warwick,
Coventry CV4 7AL, UK. ranko.lazic@dcs.warwick.ac.uk

Abstract. This paper shows how to translate the problem of deciding trace refinement between two data independent (DI) CSP processes to an unreachability problem in a DI Unity program. We cover here the straightforward but practically useful case when the specification satisfies a normality condition, **Norm**, meaning that we do not have to worry about hidden or *unrecorded*¹ data variables. This allows us to transfer results about the decidability of verification problems involving programs with data independent arrays from UNITY to CSP.

Keywords: CSP, Data independence, Trace refinement, Model checking, Reachability, Array.

1 Introduction

Algorithmic formal verification of hardware and software systems is an important way of establishing system correctness or to debugging (whichever case applies!). *State exploration* of various types is commonly employed. Whereas state exploration of control structures is usually effective and efficient, handling data is more challenging. Often the only way of including data into correctness checks is to branch it into control structure, so that two instances of a control state with different data become, in effect, different control states. This often creates enormous state explosions and thus limits the techniques' applicability. We seek to improve our understanding of the special properties of data in algorithmic verification to reduce this problem.

The problem of algorithmic verification of data-bearing systems in general is hard. But an important subclass of data-bearing systems where useful results are available is the *data independent* ones. Intuitively, data independence means that data is opaque: although it can be communicated (input and output) and stored, the only way to probe the information inside is by equality testing with

¹ Roughly speaking, unrecorded variables are DI variables assuming values internally generated (e.g., by DI replicated choice as in Section 4.1), rather than inputted from environment.

another of its kind. Thus a data independent type X is nothing more than an abstract “set”. As sets of equal size are substitutable via bijection, only the size of X matters.²

Typical examples of data independence are: types of data stored in buffers or transmitted by protocols, and types of addresses/pointers in a cache or memory (equality testing is used to check if two pointers point to the same location).

Data independence has been studied in the past from the process algebra [2] and temporal logic [3] perspectives. Our research has explored both of these [4, 5] and we have formulated a unifying semantic framework to capture data independence [6,7].

While some past work on data independence assumes the infinity of DI types[2,3], no such requirement is present in our work. Actually; our work can handle infinite families of parameterised systems, where the type X can be of arbitrary nonempty finite size or infinite. The verification problem we face is more than just the uniform verification of parameterised systems [8].

Data independence is also related to another important technique in analysing data-bearing systems: symbolic labelled transition systems [9,10]. A symbolic LTS tries to represent, in a finite graph, a system which is potentially infinite thanks to data-bearing. Variables are left as symbols instead of being instantiated by concrete values. Symbols and their properties (described in a boolean expression) are used in the system so that a set of concrete value instantiations is treated in one go instead of each concrete value instantiation in a separate go.

Other forms of symbolism (e.g., those based on binary decision diagram, regular language, regions, etc.) also flourish in model/refinement checking for overcoming problems like the tractability of large state spaces, abstraction of unbounded structures (like queues, stacks, linear integer, and parameterised linear topology), and abstraction of real time and hybrid systems. The relationship between these works and data independence has yet to be fully explored.

The overall aim of the present paper is to relate the DI work that we have done more recently on a Unity-like language to that we did previously in CSP. Besides the obvious benefits of connecting and unifying different data independence theories, there are some other very important practical ones, since it means that results obtained in the Unity style may now apply to CSP. One particular class of result we want to borrow are our recent decidability results on DI arrays[11,5,12].

Secondly, in practical applications, both approaches have relative merits and drawbacks. The benefits of our³ process algebraic framework are: unified language, compositionality, and stepwise refinement, while the benefits of a temporal logic approach are: abstract property and liveness/fairness friendliness. So it would often be very instructive if some light could be shed on how the same case study could be differently encoded and processed in different approaches.

² It is sometimes possible to weaken the definition of data independence in carefully controlled ways, such as allowing symbols representing functions from the DI type to a known finite type such as the booleans[1].

³ Some of these are specific to CSP.

And the most direct way to show that is by a translation between the checking problems of the two frameworks.

Thus, the rest of this paper will aim to give a two-step translation procedure that can, through an intermediate SLTS, automatically convert a CSP DI refinement checking problem to an equivalent Unity DI model checking problem, namely unreachability.

$$Spec \sqsubseteq_X Impl \xrightarrow{\text{translate}} Prog \models_X Unreachability$$

The reasons why we restrict the property to reachability/unreachability are

- Reachability is a powerful concept. Both trace refinement and (we believe) stable-failure refinement can be encoded into it. However it cannot be true for failure-divergence refinement because of the infinitary nature of divergence: an effective translation would imply solubility of the halting problem.
- By restricting ourselves to reachability, decidability results on more powerful classes of programs with DI arrays [11] are achievable. This extension to our reasoning power with arrays contains a large proportion of likely practical applications.
- Reachability is a natural target for our translation problem, and corresponds to the intuition that both trace properties and unreachability correspond naturally to *safety* specifications.

2 The Norm Condition and This Paper

Translating from the CSP language to a Unity fragment language is not easy, especially since there is a mismatch in the expressive power of the two languages.

The Unity fragment used in [11,5] is a much simpler language than CSP. In essence, its programs consist of a finite set of variables, an initialisation of the variables, and a finite set of guarded multiple assignments of the form, *boolean_expression* $\longrightarrow$ {*assignment*}.

Simplicity buys unreachability checking on Unity DI programs a useful property: monotonicity.

$$Prog \models_{T_1} Unreachability \wedge |T_2| \leq |T_1| \Rightarrow Prog \models_{T_2} Unreachability$$

Similar properties, however, are not enjoyed by CSP refinement checking in any of its three major models. In CSP we even can construct (*Spec*, *Impl*) pair that refines (in all three models) iff the size of *T* is an odd number (c.f. Chapter 5 in [4]). So some *Spec* $\sqsubseteq$ *Impl* are inherently untranslatable to *Prog* $\models$ *Unreachability*.

Another symptom of the language power mismatch is found in the handling of arrays.

Some CSP processes can use DI variables to simulate a limited form of DI arrays.

A good example of that is the specification of a nondeterministic register.

$$\begin{aligned}
Spec_{NR} \equiv & \text{let } NR(x) = out!x \rightarrow NR(x) \\
& \square \\
& in?y_0 \rightarrow NR(x) \sqcap NR(y_0) \\
& \text{within } in?x_0 \rightarrow NR(x_0)
\end{aligned}$$

When considering the effect of this process as the left-hand side of a refinement check, the parameter variable x above actually encodes a boolean DI array (i.e., a DI set); a more intuitive equivalent representation using explicit array variables is⁴,

$$\begin{aligned}
Spec'_{NR} \equiv & \text{let } NR(arr) = out\$x : arr[x] \rightarrow NR(\{x\}) \\
& \square \\
& in?y_0 \rightarrow NR(arr \cup \{y_0\}) \\
& \text{within } in?x_0 \rightarrow NR(\{x_0\})
\end{aligned}$$

The root of the problem with $Spec_{NR}$ is that many nondeterministic choices may be made before their effects are shown by output. In $Spec'_{NR}$ above, in order to delay the choice until its effect is seen, we need to introduce an explicit array. So $Spec_{NR}$ effectively enjoys the power of arrays even without introducing them.⁵

But this cannot be true in Unity, where it can be shown to be impossible to simulate DI arrays using simple DI variables. Actually, to translate $Spec_{NR} \sqsubseteq_X Impl_{NR}$ to Unity for some $Impl_{NR}$, $Spec_{NR}$ need be first transformed to something like $Spec'_{NR}$ in order to bring out the hidden arrays in it. This is called *normalisation*; that is, transforming a specification to a form satisfying the **Norm** condition below.

Norm

A process satisfies this if and only if

- (i) *there is no replicated choice over DI types (except for input ? and nondeterministic selection \$);*
- (ii) *no hiding or renaming;*
- (iii) *only alphabetised parallel;*
- (iv) *each branch of a multi-path choice⁶ is disjoint with the others in the set of channels it will use for the next communication.*

Norm is essentially what appears in [13], which extends the definition in [4] by allowing alphabetised parallel.

Norm and normalisation are closely related to the traditional notion of determinisation, the algebraic normal form of CSP in Chapter 11 of [13], and the

⁴ The syntax used for the example is mainly taken from $CSP_{di-SPEC}$ in Section 4.1. The only addition is the use of selective nondeterministic selection ($out\$x : arr[x]$), which is a dual of selective input (i.e., $in?x : b$) in $CSP_{di-SPEC}$.

⁵ We note that this implies that in order to translate refinement questions like that of $Spec_{NR}$ to Unity, it is likely that arrays would be required in the Unity even if not in the CSP.

⁶ Multi-path choices are the choices of form $\sqcap\{P_1, \dots, P_n\}$ and $\sqcap\{P_1, \dots, P_n\}$; binary choices are their special cases.

normal form computed by FDR [14]. To understand it fully, some explanation of nondeterminism in CSP is in order.

Nondeterminism in CSP usually means extensional nondeterminism⁷, which is defined in the extensional semantics of processes. The semantics must be fine-grained enough though, like the stable failures and failures divergences models, to be able to record that nondeterminism. It means nondeterminism in externally observable behaviour, rather than nondeterminism in the graph structure of transition systems (i.e., the nondeterminism from invisible actions and multiple actions with the same labels). We call this second sort *graph nondeterminism*.

In checking CSP processes, tools do not usually calculate their extensional semantics directly. Instead, various kinds of transition graphs are used, like plain LTSs, symbolic LTSs, or GLTSs (i.e., LTSs annotated with minimal acceptances and divergence [15]). These graphs, of course, often exhibit graph nondeterminism, and it is important that we understand this and how it relates to the extensional variety.

Extensional nondeterminism usually implies graph nondeterminism; for instance, an extensionally nondeterministic process must have a graph-nondeterministic plain LTS. But it is not absolutely so for other graphs, since the same process may have a graph-deterministic GLTS.

For the purpose of this paper, we only need to consider graph nondeterminism, since the trace model is too coarse-grained to record any extensional nondeterminism. In the rest of this paper, whenever nondeterminism, determinism, or determinisation is mentioned, it means the graphical sort. Moreover, sometimes the special CSP terminology for graph determinism, like **Norm** and normalisation⁸, are also used.

Many of these ideas are illustrated well by the $Spec_{NR}$ and $Spec'_{NR}$ examples above. The first gets its nondeterminism from the branching (implemented using internal actions) of the choice operator $\sqcap$. The second, extensionally equivalent process, gets its from the selection $\$$. The first fails **Norm** because it has the same channel on either side of the $\sqcap$, whereas the second passes it. It would be possible to eliminate all graph nondeterminism from the second by suitable labelling of the nodes where $\$$ appears; this is not possible for the first. There is a clear sense in which $Spec'_{NR}$ is the normal form of $Spec_N R$.

Going back to the language mismatch discussion above, it is now clear that normalisation and monotonicity above are the two major barriers to translating CSP refinement checks to Unity. As a matter of fact, in [16] most of the attention has been devoted to the study of the two problems. It is shown that monotonicity and normalisation are inherently related; any CSP specification that is normalisable in a defined sense will automatically possess a monotonicity property in its refinement by any implementation.

⁷ For example, when we say $Spec$ is more nondeterministic than $Impl$, we mean it extensionally.

⁸ Strictly speaking, graph-deterministic graphs only correspond to pre-normal form graphs as they may contain semantically equivalent nodes and so not be complete normal forms yet [13].

We therefore think it is important to set out the results for this case, all the more so because most natural specification processes either satisfy **Norm** already or can be altered easily to do so. By assuming that all specifications do satisfy **Norm**, this paper shows that the translation procedure (esp., that of specifications) can be presented in a much simpler way than in [16].

The paper is organised as follows. Section 3 illustrates the general idea of the translation, while Section 4 introduces the basic formalisms and definitions used. Section 5 develops details of the translation, where Section 5.1 translates *Impl* to a Unity *generator*, Section 5.2 translates *Spec* to a Unity *acceptor*, and section 5.3 connects the two to form the whole *Prog* for unreachability checking. Section 6 concludes the paper with pointers to some promising future work.

3 The General Ideas

3.1 Generators and Acceptors

The key point of the translation from CSP refinement to Unity unreachability lies in the construction of a Unity program, *Prog*, and the identification of the states in *Prog* that are required to be unreachable.

Our initial idea about a possible solution came from the refinement checking procedure used in FDR. Later, as only the trace model is treated as a first step, we realised that it can be simplified and presented in an automata-theoretical approach following [17], which is what we now present.

The idea is to construct *Prog* to implement refinement checking on (*Spec*, *Impl*) by exploiting nondeterminism in Unity: the refinement will fail if and only if *Prog* will, for some set of nondeterministic choices, reach a designated control state.

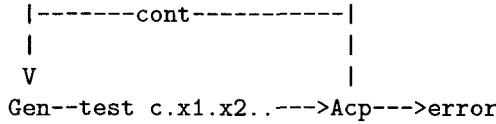
- *Impl* is implemented in Unity as a nondeterministic behaviour generator, *Gen*, which generates through nondeterminism all possible traces of *Impl* and only them. (Note that the prefix-closedness of traces means all states of the automaton are accepting.)
- *Spec* is implemented in Unity as a deterministic behaviour acceptor, *Acp*, which accepts exactly the traces of *Spec*. The responsibility of an acceptor is to identify errors in the behaviour of generators.

Basically, the idea of acceptor, like the ideas of tester process [18] and monitor automaton [19], is just another reformulation of a well-known algorithm in deciding language containment between two finite automata. The algorithm consists of two steps:

- The first step is to determinise *Spec* and calculate its complement, $\overline{Spec}$.
- The second step is to check the emptiness of the intersected language, $\overline{Spec} \cap Impl$

Normalisation and the translation to an acceptor correspond to the first step, while unreachability checking corresponds to the second. Normalisation makes sure that all acceptors will be deterministic; that is, it never refuses a correct behaviour, and always refuses any erroneous behaviour.

- Run *Gen* and *Acp* in parallel. *Gen* outputs an event and waits. *Acp* inputs it and sees if it is acceptable. If accepted, *Acp* signals *Gen* to continue. Otherwise, an *error* occurs. (Note the two automata perform the check interactively, rather than on a word-by-word basis, as they do in Section 2.5 of [20]).



Thus $Prog = Gen \parallel Acp$.

- Model checking to verify unreachability of *error* on *Prog*

$$Spec \sqsubseteq_X Impl \xRightarrow{translate} Prog \models_X Unreachability(error)$$

3.2 The Procedure

Unity is a state-based language; states are defined by value assignments on a finite set of variables. In its pure form, even control states are encoded into variables. The semantics of Unity programs (esp., DI programs) can be naturally captured by a SSTS (Symbolic State Transition System).

On the other hand, CSP is an action-based formalism. Its classic operational semantics is based on plain labelled transition systems; though for data-bearing CSP, symbolic LTSs may be more convenient from the point of view of algorithmic verification.

But the difference between SSTSs and SLTSs is not great. They can easily be translated to-and-fro just as can plain STSs and LTSs [18]. In the rest of this paper, therefore, only SLTSs will be used explicitly. Based on SLTSs, our translation procedure for implementations and specifications can be formulated as below.

$$\begin{aligned} Impl &\Rightarrow SLTS \Rightarrow Gen \\ NSpec &\Rightarrow SLTS_N \Rightarrow Acp \end{aligned}$$

NSpec is a specification satisfying **Norm**, while *SLTS_N* is a deterministic SLTS. *Acp* not only accepts the behaviours defined in *SLTS_N* but also monitors the behaviours outside it. *Gen* only generates the behaviours defined in *SLTS*, i.e., with the help of nondeterminism.

4 The Basics of Our Formalism

4.1 CSP

A thorough treatment of CSP can be found in [13]. Here we give only a brief introduction to the dialects of the CSP language used in this paper. To help with the intuition of acceptors/generators and to make explicit (at least partially) the

fact that specifications are constrained by **Norm**, we use two different dialects: $CSP_{di-IMPL}$ for implementations and $CSP_{di-SPEC}$ for specifications.

Syntax of $CSP_{di-IMPL}$

$$\begin{aligned}
 LLP &::= l(e_1, \dots, e_n) \\
 &\quad | \sqcap x : X \bullet LLP \mid \sqcap \{LLP_1, \dots, LLP_n\} \\
 &\quad | c!x_1 \dots !x_n \rightarrow LLP \mid b \ \& \ LLP \\
 &\quad | \text{let } l(z_1, \dots, z_n) = LLP' \text{ within } LLP
 \end{aligned}$$

$$P ::= LLP \mid P \parallel [chans] P' \mid P \setminus chans$$

Syntax of $CSP_{di-SPEC}$

$$\begin{aligned}
 LLQ &::= l(e_1, \dots, e_n) \\
 &\quad | \sqcap \{LLQ_1, \dots, LLQ_n\} \\
 &\quad | c?x_1 \dots ?x_n : b \rightarrow LLQ \mid b \ \& \ LLQ \\
 &\quad | \text{let } l(z_1, \dots, z_n) = LLQ' \text{ within } LLQ
 \end{aligned}$$

$$Q ::= LLQ \mid Q \parallel [chans1 \mid chans2] Q'$$

where *chans* is a set of channel names, e.g., $\{c_1, \dots, c_n\}$, and

$$\begin{aligned}
 b &::= true \mid false \mid x = x' \mid arr[x] \mid \neg b \mid b \wedge b' \mid b \vee b' \\
 e &::= x \mid are \\
 are &::= arr \mid \{ \} \mid \{x\} \mid X \mid are \cup are' \mid are \cap are' \mid are \setminus are'
 \end{aligned}$$

In the above, a data independent type X is assumed, and x, y, x_i and y_i range over the set (i.e., χ) of variables of type X . arr and arr_i range over the set (i.e., ARR) of variables of boolean DI array type, z and z_i range over the set of variables of both types, i.e., range over $Z = \chi \cup ARR$. A value expression (e) can either be a DI variable (x) or a DI array value expression (are). Value expressions can only be used in a function call ($l(e_1, \dots, e_n)$) as actual parameters. A function (l) is defined using the **let** construct. In general, it is assumed that each variable's binding occurrence must be unique in a process expression.

$CSP_{di-IMPL}$ is designed to capture the intuition of pro-active and “talkative” generators; all the choice is made internally and processes only output. As there is no **Norm** requirement on implementations, $CSP_{di-IMPL}$ can enjoy the full power of interface parallel and hiding operators. $CSP_{di-SPEC}$ is designed to capture the intuition of passive and receptive acceptors, so *multipath choice* ($\sqcap \{LLP_1, \dots, LLP_n\}$ or $\sqcap \{LLQ_1, \dots, LLQ_n\}$) is made externally and it only inputs. Its higher level operators are confined by **Norm** to alphabetised parallel. (This can be expressed in terms of the interface parallel operator used in $CSP_{di-IMPL}$, but unlike interface parallel cannot introduce graph nondeterminism.) Other features of the languages are:

- The languages implement the intuition of “communicating sequential process”, where a system (i.e., P or Q) consists of a network of sequential processes (i.e., $LLPs$ or $LLQs$) running in parallel. Only low-level operators [15] can be used in LLP and LLQ definition. High-level operators like parallel and hiding are used only in composing the network. It will ensure the finite-control (i.e., finite control states) of any process in the language.
- Replicated internal choice in $CSP_{di-IMPL}$ is used to capture the intuition of *internal* or *unrecorded* variables, which cause (sometimes insuperable) problems to the process of translation to an acceptor if they appear on the left hand side of a refinement check. In the current paper, however, this possibility is banned by (i) of **Norm**.
- Selective input in $CSP_{di-SPEC}$ helps introduce *external variables*, or *recorded variables*, explicitly; that is, variables with values assigned by the environment and therefore “recorded” in its behaviours.
- *Boolean guard* (b & LLP or b & LLQ) and multipath choice are used in place of conditionals for the sake of expressiveness and simplicity.
- The condition b in boolean guards is the conjunction, disjunction and negation of two most basic forms of testing: equality testing ($x = x'$) and array testing ($arr[x]$).

Syntactically, the two languages look very restricted and very different from each other. But (trace-)semantically they are, actually, quite expressive and very close to each other.

- Output can be simulated by selective input in $CSP_{di-SPEC}$. Input can be simulated by output and replicated internal choice (given that we use the trace model) in $CSP_{di-IMPL}$.
- Any fixed-finite data types and their operations can be reduced by branching and instantiation into control structure (c.f. [21] for an actual procedure, where case analysis and mutual recursion are used to reduce value-passing CCS to pure CCS). Although this is not recommended for doing real model checking, it is absolutely legitimate and simplifying when developing the theory of data independence and what is formally decidable.
- A finite collection of DI arrays with contents of fixed-finite types and *map* operations (i.e. mapping a n -ary fixed-finite operation onto a set of n -tuples which collect their n fixed-finite elements from the same location of n DI arrays [11]) can be reduced to a number of sets (i.e. boolean arrays) and combinations of the 3 basic set operations ($\cup$, $\cap$ and complementation).

4.2 Unity

As explained earlier, a Unity program consists of a finite set of variables, their initialisation, and a finite set of guarded commands.

Unity variables are typed. For programs in this paper, besides the DI type and the boolean DI array type of the CSP languages, some fixed-finite types are also allowed to encode the control structures and synchronous communications in CSP, since Unity language itself supports neither.

For control structures, a control variable, CS , encodes the control states of a CSP process, which will correspond one-to-one to each node in $SLTSs/SLTS_Ns$. For communication, a channel variable, CN , is used to record the channel name on which the communication occurs, and a binary flag variable f is used to implement the synchronisation between processes. The report of error by an acceptor after encountering an illegal traces is made through a binary variable, r .

The set of assignments in each guarded command are simultaneous. That is, the RHS expressions of the assignments are evaluated first; then the evaluated values are assigned to the LHS variables all at once. If we temporarily ignore the fixed finite types and their operations⁹, the formal definition of guarded commands can be given below,

$$cmd ::= b \longrightarrow as^{10}$$

where b is a boolean guard as in CSP and $as ::= arr := are \mid x := x' \mid x := ?$. The only construct we need to pay attention to is nondeterministic selection, $x := ?$. It means to pick nondeterministically a value from X and assign it to x . It is a form of data nondeterminism. With it we can generate DI values implicitly.

After instantiating X with a concrete type T , a Unity DI program becomes an ordinary Unity program, whose semantics is modelled by a concrete state-based system. Each concrete state of the system is identified by a value assignment on the set of program variables. Initialisation is the value assignments identifying the initial (concrete) states $\{s_0^1, s_0^2, \dots, s_0^k, \dots\}$. The dynamics of a Unity program can be understood through the notion of *runs*.

A run is a finite sequence of (concrete) state transitions starting with a initial state s_0 ,

$$s_0 \xrightarrow{cmd_1} s_1 \xrightarrow{cmd_2} s_2 \dots \xrightarrow{cmd_n} s_n$$

That is, for any state s_{i-1} ($1 \leq i \leq n$), if the guard of cmd_i is true, and cmd_i is fired, it will transit to state s_i . (n can be 0, in which case a run degenerates to a single initial state.)

A Unity program is a closed system; it is graph-deterministic iff it has only one run. That is quite simple, but not very useful. More commonly, we will study the deterministic subsystems of a Unity program.

Definition 1. *A subprogram A of a Unity program S is deterministic iff at any point of any run of S , the subset of commands belonging to A has at most one member enabled, and the member must not use nondeterministic selection in its assignments.*

So caution should be taken on what determinism it means when we mention a deterministic Unity program.

⁹ Fixed-finite types and their predicates/operations are theoretically trivial, although their treatment can clutter our presentation non-trivially.

¹⁰ Note in this context we use vectors interchangeably with sets.

One important property of Unity DI programs is the monotonicity of unreachability with respect to the size of X . It is quite obvious: by increasing the size of X , nondeterministic selection can pick more values, so it can simulate all the runs of the program with a smaller X instantiation, in particular ones reaching designated control states.

More interestingly, that also implies the monotonicity of determinism in Unity. So we need only to check the determinism of a Unity (sub)program when X is infinite to make sure it will be uniformly deterministic over all possible instantiations of X . Generally, when we mention a Unity DI (sub)program is deterministic, we mean it uniformly.

4.3 The Symbolic LTS

A SLTS is a data-bearing LTS¹¹. For DI systems, it adds to a LTS the following,

- Each node is associated with a set of data variables of DI type or boolean DI array type. $vars(n)$ denotes the set of variables for node n .
- Each transition is labelled by a triple of *guard*, *symbolic event*, and *assignments*: $ts ::= (gu, se, as)$. Below is a transition from m to n .

$$m \xrightarrow{gu, se, as} n$$

- Free and bound variables in the label of a transition must satisfy the following constraint to make the overall SLTS well-formed:

$$\begin{aligned} &fv(se) \cup fv(as) \subseteq vars(m) \wedge fv(gu) \subseteq vars(m) \cup bv(se) \\ &\wedge vars(n) \subseteq vars(m) \cup bv(se) \cup bv(as) \end{aligned}$$

The possible symbolic events (se), guards (gu), and assignments (as) are:

$$\begin{aligned} se &::= c \ tt \mid \tau \quad (\ tt ::= \epsilon \mid ?x \ tt \mid !x \ tt) \\ gu &::= true \mid false \mid x = x' \mid arr[x] \mid \neg gu \mid gu \wedge gu \mid gu \vee gu \\ as &::= arr := are \mid x := x' \mid x := ? \end{aligned}$$

The x in $?x$ of se , and the x (or arr) in $x := x'$ (or $arr := are$) of as are binding occurrences. All other occurrences of DI or array variables in a label are free occurrences.

The intuitive meaning of a transition $m \xrightarrow{gu, se, as} n$ is,

If the values of variables in node m and the values of input variables (bound variables) in se satisfy gu , the transition is able to fire, after which the evaluated RHSs of as are simultaneously assigned to the (old or new) LHS variables.

¹¹ For each process, it is assumed that its LTS or SLTS has a unique *root* node.

Note that as input variables in se can be constrained in gu , SLTS transitions can easily encode the selective input of CSP. One complication it might bring us, however, is: there might be name conflicts in gu , since it is possible that $vars(m)$ and $bv(se)$ might intersect. We need to adopt a convention that whenever a name conflict arises, e.g., on variable x , the primed variable x' will be used to refer to the x in $bv(se)$. More details on this will be in Section 5.2.1.

An SLTS after instantiating X with a concrete type T becomes a concrete SLTS. Concrete states of such SLTSs consist of two parts: a node name, e.g., m , identifying the control state, and a value assignment on $vars(m)$, identifying the data state. A sequence of concrete states connected by concrete transition labels form a run of these SLTSs, where a concrete transition label is a transition label (ts) with its symbolic event (se) replaced by a concrete event.

A concrete SLTS is a concrete communicating system; its determinism is based on traces, which is a sequence of concrete events implied by a run. Each run implies a trace, which is just the sequence of concrete events in the (concrete) transition labels of the run. Whenever a run implies a trace, we say the run conforms to the trace.

Definition 2. *A concrete SLTS is deterministic iff, for any trace tr of the SLTS, the run conforming to it must be unique and does not use any non-deterministic selection in the transition labels.*

Based on a similar argument as in DI Unity, the monotonicity of determinism in SLTSs can also be shown to be true. So when we say a SLTS is deterministic, we mean it uniformly.

In general, it is difficult to devise a complete algorithm to decide the determinism of SLTSs (or Unity DI (sub)programs). But an easy-to-check sufficient condition can satisfy most of our needs.

Lemma 1. *A SLTS is deterministic, if the transition labels in the graph use no τ event or nondeterministic selection, and the sibling transitions either share no channel or are disjoint on their guards.*

A SLTS satisfying the condition is also called normalised, or a $SLTS_N$.

5 The Translation

In CSP, a process consists of a network of sequential *LLPs* running in parallel. To translate it from CSP to Unity, we adopt a compositional approach.

Firstly, each component *LLP* is translated to a basic automaton in Unity. Then, these basic automata are composed up by a Unity simulation of CSP parallel and hiding operators. This allows us to create the generator and acceptor we needed.

5.1 *Impl to Gen*

Translating the implementation to a generator is relatively straightforward. We simply follow the procedure outlined below.

For each component LLP in *Impl*, do the following.

Step 1: $LLP \Rightarrow SLTS$

Symbolic Labelled Transition Rules

$$\begin{array}{l}
 c!x \rightarrow LLP \xrightarrow{true, c!x, \{\}} LLP \\
 l(e) \xrightarrow{true, \tau, z:=e} LLP \quad \text{where } l(z) = LLP \\
 \Box x : X \bullet LLP \xrightarrow{true, \tau, \{x:=?\}} LLP \\
 \Box \{LLP_1, \dots, LLP_n\} \xrightarrow{true, \tau, \{\}} LLP_i \\
 b \ \& \ LLP \xrightarrow{b, \tau, \{\}} LLP
 \end{array}$$

Each node n is identified by a process expression LLP . The set of variables associated with n is, $vars(n) = fv(LLP)$.

Step 2: $SLTS \Rightarrow Gen$

* *Interface with the environment*

To implement the SLTS in Unity, first thing to do is to model communication by shared variables, so *Gen* uses a set of *interface variables* to communicate with the environment. Specifically, the following interface variables need be defined, $CN : \{c_1, \dots, c_n\}$ for the channel name, $DC : seq X$ for the vector of data components, and $f : \{cont, test\}$ for a synchronisation flag.

* *Encode control and data*

The control structure of the SLTS needs to be encoded by a control state variable, $CS : \{root, m, n, \dots\}$, ranging over the node names of the SLTS. Data variables associated with nodes in the SLTS should accordingly be implemented as Unity data variables of the same type. All the control and data variables are *private variables* of *Gen*.

```

-----
| f | CN | DCs |
|-----|
|               |
|  Private variables  |
|-----|
    
```

* *Implement transitions*

$$m \xrightarrow{gu, \tau, as} n \quad \text{by} \quad CS = m \wedge gu \longrightarrow \{CS := n\} \cup as$$

$$m \xrightarrow{gu, c!x, as} n \text{ by} \\ CS = m \wedge gu \wedge f = cont \longrightarrow \{CS := n, CN := c, DC := x, f := test\} \cup as$$

* *Initial states*

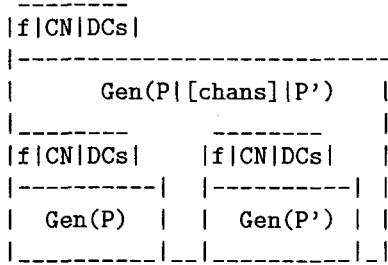
$$f := cont \quad CS := root$$

The implemented *Gen* will be able to generate all the possible traces of the SLTS nondeterministically, in the sense that the value of each communication appears in *CN* and *DC*, and separate events are identified by changes in *f*. A new event is recorded when *f*'s value is changed by the process from *cont* to *test* (it being the duty of the observer to change it the other way once the event has been observed).

Hence, *LLPs* can be translated into basic sequential *Gens*. Based on these results, we can continue to translate *Impl*, which is a network of *LLPs*, into a composite *Gen*.

Case 1: Parallel operator

$P \parallel chans \parallel P'$, where *P* and *P'* have been implemented as *Gen(P)* and *Gen(P')*.



* *Commands*

The following three guarded commands observe the external variables of the two subprocesses and combine/transmit these to appear in the external variables of the combination.

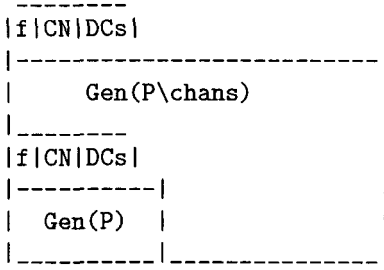
$$\begin{aligned}
& f = cont \wedge f_P = test \wedge CN_P \notin chans \\
& \quad \longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f := test\} \\
& f = cont \wedge f_{P'} = test \wedge CN_{P'} \notin chans \\
& \quad \longrightarrow \{CN := CN_{P'}, DC := DC_{P'}, f_{P'} := cont, f := test\} \\
& f = cont \wedge f_P = test \wedge f_{P'} = test \wedge CN_P = CN_{P'} \wedge CN_P \in chans \wedge DC_P = DC_{P'} \\
& \quad \longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f_{P'} := cont, f := test\}
\end{aligned}$$

* *Initial states*

$$f := cont$$

Case 2: Hiding operator

$P \setminus chans$, where *P* has been implemented as *Gen(P)*.



* *Commands*

The following either transmit or conceal each action of $Gen(P)$ as appropriate.

$$\begin{aligned}
 f_P &= test \wedge CN_P \in chans \longrightarrow \{f_P := cont\} \\
 f &= cont \wedge f_P = test \wedge CN_P \notin chans \\
 &\longrightarrow \{CN := CN_P, DC := DC_P, f_P := cont, f := test\}
 \end{aligned}$$

* *Initial states*

$$f := cont$$

5.2 Spec to Acceptor

This is similar to the translation from *Impl* to a generator. The various cases are given below.

For each component LLQ in $NSpec$, do the following.

Step 1: $LLQ \implies SLTS_N$

Due to the determinism requirement on $SLTS_N$ s, the symbolic labelled transition rules for specifications need to have one important difference from those of implementations; that is, no τ event can be generated by the rules. Therefore, the specification rules will be designed to merge all possible τ transitions with their subsequent visible ones.

Symbolic Labelled Transition Rules

$$\begin{array}{c}
 \frac{c?x : b \rightarrow LLQ \quad b[x'/x], c?x', \{\}}{c?x : b \rightarrow LLQ} \\
 \frac{LLQ_i \xrightarrow{gu, se, as} LLQ'_i}{\square\{LLQ_1, \dots, LLQ_n\} \xrightarrow{gu, se, as} LLQ'_i} \\
 \frac{LLQ \xrightarrow{gu, se, as} LLQ'}{b \& LLQ \xrightarrow{gu \wedge b, se, as} LLQ'} \\
 \frac{LLQ \xrightarrow{gu, se, as} LLQ'}{l(e) \xrightarrow{gu[e/z], se[e/z], as[e/z] \cup x' := e} LLQ'} \quad \text{where } l(z) = LLQ
 \end{array}$$

The middle two rules of symbolic labelled transitions are quite obvious. The first and last rules need to assume a set χ' of primed variables for χ and a set ARR' of primed variables for ARR so that for any $x \in \chi$, there is a corresponding

$x' \in \chi'$, and similarly for any arr . In the context of a transition $\xrightarrow{gu, se, as}$, all occurrences of new variables introduced by binders in se and as are renamed to their primed counterparts to avoid name conflicts. Note that due to the unique binding occurrence condition on a process expression, only the variables going out of scope through a (direct or indirect) recursive function call may conflict with new variables. The proper working of this mechanism also depends on **Norm** of LLQ , which guarantees that all recursive function calls are either action-guarded or are calling on (recursive) functions that behave like *STOP* or *DIV*.

Each node n in the resulting $SLTS_N$ is identified by a process expression LLQ . The set of variables associated with n is, $vars(n) = fv(LLQ)$. No sibling transitions in the $SLTS_N$ share any communication channel.

Step 2: $SLTS_N \implies Acp$

* *Interface with the environment*

As in *Gen*, the interface includes $CN : \{c_1, \dots, c_n\}$ for the channel name, $DC : seq X$ for the vector of data components, and $f : \{cont, test\}$ for synchronisation. Moreover, *Acp* needs an additional variable to report error in accepting, $r : \{normal, error\}$.

* *Encode control and data*

Also like *Gen*, we need a control state variable, $CS : \{root, m, n, \dots, stop\}$, bookkeeping the current node in execution or simply *stop*, and a set of data variables implementing variables associated with each node in the $SLTS_N$. They are all private variables of *Acp*.

* *Implement transitions*

Transition $m \xrightarrow{gu, c?x', as} n$ is translated to two commands:

$$\begin{aligned} CS = m \wedge f = test \wedge CN = c \wedge gu[DC/x'] &\longrightarrow \{CS := n, x := DC, f := cont\} \cup as \\ CS = m \wedge f = test \wedge CN = c \wedge (\neg gu[DC/x']) &\longrightarrow \{r := error, CS := stop\} \end{aligned}$$

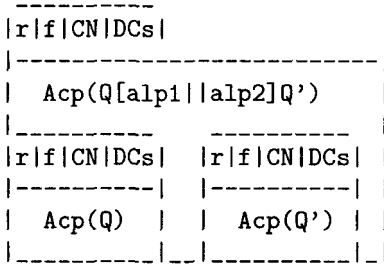
The above raises an error flag if illegal data arrives along a channel that state m can perform. (We note that the conditions of **Norm** ensure that there is at most one transition of m labelled by each channel c .) The following does the same if a communication along a channel appears with no transitions in m .

$$CS = m \wedge f = test \wedge CN \notin Chans(m) \longrightarrow \{r := error, CS := stop\}$$

* *Initial states*

$$f := cont \quad r := normal \quad CS := root$$

After we translate $LLQs$ to $Acps$, only one step is needed to translate *Spec*, that is, translating alphabetised parallel operator.



* A new private variable $ST : \{testing, resting, stop\}$ and six commands in Unity

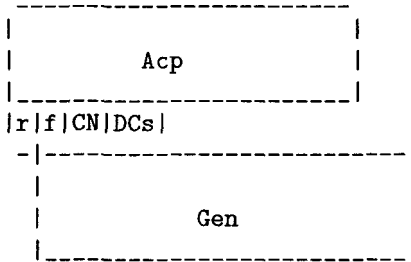
$$\begin{aligned}
 & f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp1 \cap alp2) \longrightarrow \\
 & \{CN_Q.DC_Q := CN.DC, CN_{Q'}.DC_{Q'} := CN.DC, f_Q := test, f_{Q'} := test, ST := testing\} \\
 & f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp1 \setminus alp2) \longrightarrow \\
 & \{CN_Q.DC_Q := CN.DC, f_Q := test, ST := testing\} \\
 & f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \in (alp2 \setminus alp1) \longrightarrow \\
 & \{CN_{Q'}.DC_{Q'} := CN.DC, f_{Q'} := test, ST := testing\} \\
 & f = test \wedge ST = resting \wedge f_Q = cont \wedge f_{Q'} = cont \wedge CN \notin (alp2 \cup alp1) \longrightarrow \\
 & \{r := error, ST := stop\} \\
 & f = test \wedge ST = testing \wedge f_Q = cont \wedge f_{Q'} = cont \wedge r_Q = normal \wedge r_{Q'} = normal \longrightarrow \\
 & \{f := cont, ST := resting\} \\
 & f = test \wedge ST = testing \wedge (r_Q = error \vee r_{Q'} = error) \longrightarrow \{r := error, ST := stop\}
 \end{aligned}$$

* Initial states:

$$f := cont \quad ST := resting \quad r := normal$$

5.3 Connecting the Generator to the Acceptor

Finally, by letting *Gen* and *Acp* sharing the common variables in their interfaces, we connect them up and obtain the *Prog* needed for unreachability check. Variable r is left open to report error.



This completes our translation of the problem, $NSpec \sqsubseteq_X Impl$, to another problem, $Prog \models_X Unreachability(error)$. Based on the translation, Theorem 4 in [11] can be transferred to the setting of CSP with arrays.

Theorem 1. *For any specification of $CSP_{di-SPEC}$ satisfying **Norm**, the problem of its trace refinement checking by any implementations in $CSP_{di-IMPL}$ is decidable.*

6 Conclusion and Future Work

We have shown how to translate certain forms of CSP trace refinement check in a syntax that allows DI array operations into Unity unreachability. With the exception of the *renaming* operator, which we have excluded for simplicity, it is possible to convert any CSP process description of the type usually run on FDR (i.e., parallel/hiding/renaming combinations of sequential processes) to a generator, after noting the trace equivalence of internal and external choice. As a substantial majority of specifications used with FDR either satisfy **Norm** or can be trivially modified to do so – for **Norm** essentially corresponds to clarity of a specification- this means we can confidently expect that our decidability results will cover many practically important cases of CSP checks involving arrays.

Nevertheless, understanding which non-normalised specification processes are capable of being transformed to a finite $SLTS_N$ is important because it will determine the extent to which more general problems of trace refinement can be decided by our methods. The main problem in doing this is understanding the role of unrecorded variables and their relationship to monotonicity. We have investigated this in [16] and introduced the concept of *DI-explicitness* as a way of understanding this. More work is required here.

In general, we believe that the methods described in this paper can be used to extend our results to stable-failures equivalence. Likewise, the work could also be extended to cover the cases of DI arrays that are indexed by one DI type (e.g., X) and containing contents of another (e.g., Y), or even the cases of multi-dimensional arrays.

The decidability results in this paper are theoretical and sometimes rely on the decision procedures arising from well-structured transition systems [22]. Finding cases where simpler ideas such as threshold calculation [4] will work is important, as is a general investigation of how to translate the decision problems from being theoretically soluble to having access to practical tools that solve them.

Acknowledgements. This work was funded by the EPSRC standard research grant ‘Exploiting data independence’, GR/M32900. Bill Roscoe and Xu Wang were partially supported by a grant from US ONR. Ranko Lazic is also affiliated to the Mathematical Institute, Belgrade, and supported in part by a grant from the Intel Corporation. Xu Wang is now supported by EPSRC grant GR/S11091/01 & GR/S11084/01 and by School of Computer Science, University of Birmingham.

References

1. Lazić, R.S., Roscoe, A.: Data independence with generalised predicate symbols. In: International Conference on Parallel and Distributed Processing Techniques and Applications. Volume I, Las Vegas, Nevada, USA, CSREA (1999) 319–325
2. Jonsson, B., Parrow, J.: Deciding bisimulation equivalences for a class of non-finite-state programs. In: Symposium on Theoretical Aspects of Computer Science. (1989) 421–433
3. Wolper, P.: Expressing interesting properties of programs in propositional temporal logic. In: Proceedings of the 13th ACM Symposium on Principles of Programming Languages. (1986) 184–193
4. Lazić, R.: A Semantic Study of Data Independence with Applications to Model Checking. PhD thesis, Oxford University Computing Laboratory (1999)
5. Lazić, R., Newcomb, T., Roscoe, A.: On model checking data-independent systems with arrays without reset. Technical Report RR-02-02, Oxford University Computing Laboratory (2002) To appear in the Journal of Theory and Practice of Logic Programming.
6. Lazić, R., Nowak, D.: A unifying approach to data independence. In: Proceedings of the 11th International Conference on Concurrency Theory. Volume 1877 of Lecture Notes in Computer Science., Springer-Verlag (2000) 581–595
7. Lazić, R., Nowak, D.: On a semantic definition of data independence. In: Proceedings of the 6th International Conference on Typed Lambda Calculi and Applications. Volume 2701 of Lecture Notes in Computer Science., Springer-Verlag (2003) 226–240
8. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. *Theoretical Computer Science* **256** (2001) 93–112
9. Hennessy, M., Lin, H.: Symbolic bisimulations. *Theoretical Computer Science* **138** (1995) 353–389
10. Lin, H.: Symbolic transition graph with assignment. In: Proceedings of the 7th International Conference on Concurrency Theory. Volume 1119 of Lecture Notes in Computer Science., Springer-Verlag (1996) 50–65
11. Roscoe, A., Lazić, R.: What can you decide about resettable arrays? (preliminary version). In: Proceedings of the 2nd International Workshop on Verification and Computational Logic, Technical Report, Department of Electronics and Computer Science, University of Southampton, UK (2001)
12. Newcomb, T.: Model Checking Data-Independent Systems With Arrays. PhD thesis, Oxford University Computing Laboratory (2003) To appear.
13. Roscoe, A.: The Theory and Practice of Concurrency. Prentice-Hall (1998)
14. Formal Systems (Europe) Ltd: Failures-Divergence Refinement: FDR2 User Manual. (1999) *<http://www.formal.demon.co.uk>*
15. Roscoe, A.W.: Model-checking CSP. In Roscoe, A.W., ed.: A Classical Mind: Essays in Honour of C.A.R. Hoare. Prentice Hall (1994) 353–378
16. Wang, X., Roscoe, A., Lazić, R.: Translating CSP trace refinement to Unity unreachability: a study in data independence. Technical Report RR-03-08, Oxford University Computing Laboratory (2003)
17. Vardi, M., Wolper, P.: An automata-theoretic approach to automatic program verification (preliminary report). In: Proceedings of the 1st Annual IEEE Symposium on Logic in Computer Science, Washington, DC (1986) 332–344
18. Valmari, A.: The state explosion problem. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models. Volume 1491 of Lecture Notes in Computer Science. Springer-Verlag (1998) 429–528

19. Alur, R., Henzinger, T.: Computer-aided verification: An introduction to model building and model checking for concurrent systems. Draft (1999)
20. Milner, R.: Communicating and Mobile Systems: the π -calculus. Cambridge University Press (1999)
21. Milner, R.: Communication and Concurrency. Prentice-Hall (1989)
22. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theoretical Computer Science **256** (2001) 63–92

Linking CSP-OZ with UML and Java: A Case Study*

Michael Möller, Ernst-Rüdiger Olderog, Holger Rasch, and Heike Wehrheim

Department of Computing Science
University of Oldenburg
26111 Oldenburg, Germany

{michael.moeller,olderog,rasch,wehrheim}@informatik.uni-oldenburg.de

Abstract. We describe how CSP-OZ, an integrated formal method combining the process algebra CSP with the specification language Object-Z, can be linked to standard software engineering languages, viz. UML and Java. Our aim is to generate a significant part of the CSP-OZ specification from an initially developed UML model using a UML profile for CSP-OZ, and afterwards transform the formal specification into assertions written in the Java Modelling Language JML complemented by *CSP_{jassda}*. The intermediate CSP-OZ specification serves to verify correctness of the UML model, and the assertions control at runtime the adherence of a Java implementation to these formal requirements. We explain this approach using the case study of a “holonic manufacturing system” in which coordination of transportation and processing is distributed among stores, machine tools and agents without central control.

Keywords. CSP, Object-Z, UML, Java, assertions, runtime checking

1 Introduction

Object-oriented (OO) design and programming languages play a major role in software development. The Unified Modeling Language (UML) [38,33] is an industrially accepted OO-modelling and design language; Java [15] is a widely used modern OO-programming language. UML and Java are thus likely to be used together in an object-oriented system development.

While UML and Java are well suited for modelling and implementation, they fall back as far as *correctness* issues are concerned. One of the main criticisms against UML is its lack of precision; Java programs are difficult to formally analyse even for experts [1,21]. Hence, with respect to reliability, a UML and Java based software development would gain from being complemented with a *formal* approach to system design. Since UML with its various diagram types allows for a multi-view modelling of a system, a formal method with the ability of specifying different aspects of a system as well is needed here. For this purpose we took CSP-OZ [12], an *integrated* formal method combining a state-based

* This research was partially supported by the DFG project ForMooS (grant OI/98-3).

specification language (Object-Z [9]) with a behaviour-oriented language (CSP [19,32]). Properties of CSP-OZ can be formally verified, for instance by applying the FDR model checker to the process semantics of CSP-OZ [14,39].

Viewed from the formal method's side, the advantage of a combination with the UML is the possibility of *graphically* specifying the object-oriented and behavioural features of a system, without having to use the less intuitive notations offered by the formal method. The formal specification is then (partially) obtained from the UML diagrams by means of a translation. This may help in gaining acceptance of the use of a formal specification language.

For the software engineering side, the purpose of integrating a formal method into the development process is twofold: on the one hand it is used for supplying UML diagrams with a precise meaning (thus opening the possibility for *verifying* a design); on the other hand it serves as a bridge between the high-level graphical UML model and the final implementation. To preserve the precision of the formal specification in the implementation we take a pragmatic approach: Java programs are annotated with correctness *assertions* using the Java Modeling Language (JML) [24] and CSP_{jassda} [27]. JML offers static assertions like pre- and postconditions and invariants for methods as well as model variables for data abstraction. CSP_{jassda} complements this by offering trace assertions in a CSP-like notation for specifying the required order of method calls. Both kinds of assertions are generated from the formal specification to ensure a tight correspondence. We stipulate that the final Java implementation is hand-written but checked at runtime against these assertions. Thus our approach involves the three levels shown in Fig. 1 (plus the level of Java programs not discussed here).¹

This approach to system development is tailored to *reactive systems*, the application domain of CSP-OZ. We have therefore chosen a specific interpretation for UML, which fits this domain best. The subset of UML used so far includes class diagrams, state machines, and the structure diagrams of UML-RT [36]. To support the combination with CSP-OZ we are developing a UML profile (with tool support), which provides specific stereotypes and tags for CSP-OZ classes and their ingredients as well as the capsules, protocols and ports of UML-RT structure diagrams. During specification generation every UML class is translated to a CSP-OZ class in which the attribute and method names are extracted from the class diagram, and the CSP part is obtained by a translation of the associated state machine. The structure diagrams describe the architecture of the system; they are translated into a CSP system description involving parallel composition [13]. In the next step, this CSP-OZ specification is used to generate JML and CSP_{jassda} specifications. The final hand-written Java program is checked against the assertions using a runtime-checker for JML [25] and the tool jassda [4,5].

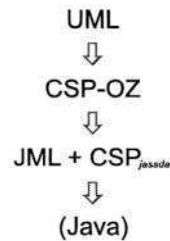


Fig. 1. Development levels

¹ There are parallels to Model Driver Architecture (MDA): Our approach involves platform independent models (PIMs) and platform specific models (PSMs) as well as transformations between them. The UML level is the main level for development.

In this paper we illustrate this approach with the case study “Holonc manufacturing system”. The case study originates from the area of production automation. In a holonic manufacturing system (HMS) autonomous transportation agents are responsible for the flow of material in a plant. Coordination of transportation and processing is distributed among stores, machine tools and agents without central control. The case study thus falls into the category of reactive systems involving a high degree of parallelism and communication.

The paper is structured as follows: according to the three levels of development the next three sections use parts of the UML model, the CSP-OZ specification and the JML/CSP_{jassda} assertions taken from the full case study to explain the different levels and transformations. It concludes with a discussion of some related work.

2 Modelling

In this section parts of the UML model of the case study are presented. A special profile, which is described next, is used to integrate CSP-OZ with the UML. After the profile the actual model parts are presented, followed by comments on the generation of CSP-OZ definitions leading to the CSP-OZ specification in Sec. 3.

2.1 A UML Profile for CSP-OZ

The UML² contains an extension mechanism in form of so-called *profiles*. They consist of *stereotype* and *tag* definitions, and *constraints* concerning the newly introduced items. It is a conservative extension mechanism, in the sense that it only allows for customisation and extension of existing model elements for specific purposes, but must not conflict with standard UML semantics. The stereotypes introduce ‘new’ model elements, with additional features represented by tags, and constraints to specialise the semantics.

The profile presented here draws on the ideas originally presented with the ROOM method [35] and modified for use with the UML under the name UML-RT in [16,36], especially those ideas dealing with reactivity, concurrency and distribution. The purpose of the profile is to provide model elements which are suited to modelling of reactive systems *and* close enough to CSP-OZ, so that an automatic generation of CSP-OZ specifications from the model is possible – provided that the modeller does not use extra-profile elements.

The main elements are *capsules*, *ports* and *protocols*. A capsule (‘actor’ in ROOM) represents a self contained, active unit of computation. It has attributes and methods almost like a typical class of an OO-language, but they are *never* accessible from the outside. Instead of method invocation used with classes, capsules are used with message passing. Message directions and types are specified in protocols; these protocols are referenced from the capsules by their ports. Compatible ports of capsules can be connected to enable communication between them. The architecture of a concrete system is specified using *structure*

² In this paper we refer to the current official version 1.5 [38].

diagrams, showing instances of capsules and the connections between their ports. A special type of visualisation is used for these diagrams (see Fig. 3), drawing the ports as small boxes on the border of a capsule and lines between ports to denote communication paths.

In contrast to ROOM/UML-RT we do not use a state machine to control the operation of a capsule by receiving messages and performing actions (internal methods calls, direct modification of the state space, sending messages), but have a strict binding of methods to ports/protocols. We use a state machine for a capsule only as a *protocol* state machine (see Fig. 4), specifying the allowed sequence of communications (with a blocking semantics) without accessing the state space.³

All subsequent stereotype and tag definitions are contained in a stereotyped UML package named «**profile**» CSP-OZ, i.e., a profile definition. Here only the main items are presented in detail. Examples are shown in Fig. 2.

Capsules. A capsule is the main building block of distributed systems. It encapsulates the state and the only way to interact with a capsule is by communication over one of its ports. Capsules can be nested to build complex capsules from simple ones, but this is not visible to the outside; likewise, the surrounding capsule has no special access to the contained capsules – all interaction has to be done via ports.

In the profile the «**capsule**» stereotype is defined for model elements of type Class. The following constraints apply: A capsule has neither public attributes nor public methods. Only the ports, i.e., the associations with a protocol, stereotyped as «**base**» or «**conjugated**», are visible outside. Methods of a capsule are either stereotyped as «**base**» or «**conjugated**» matching the stereotype of the port which references a protocol containing the signature for the method. A capsule may only inherit from capsules. This stereotype has the following tags: *invariant* holds the Z predicate for the class invariant; *init* contains a Z predicate describing the initial state.

Protocols. A protocol is used to define a binary communication pattern. It distinguishes two sides of the communication, namely, *base* and *conjugated*. The communication pattern is specified from the view of the base role. From the CSP-OZ point of view, a protocol describes a number of channels (one for each operation defined by the protocol) by specifying their channel types. Protocols are primarily used to aid graphical modelling on the UML level and are only represented by definitions of communication channels for their operations in the CSP-OZ classes.

In the profile the «**protocol**» stereotype is defined for model elements of type Class. The following constraints apply: A protocol has no attributes. It is an abstract class that has no methods, but only operations, which are stereotyped either as «**in**» or «**out**». A protocol may only inherit from protocols.

³ Protocol state machines (enhanced with a blocking semantics) fit very well to the CSP process expressions of CSP-OZ classes.

Ports. Ports are represented as references to protocols. A port is just a concept of modelling complex (typed) communication endpoints and does not appear as an attribute of a CSP-OZ class. Two ports can be connected (on the structure level specifying a concrete instance of a system) only, if they both reference the same protocol *and* one is a base port and the other is a conjugated port. For a conjugated port «in» and «out» decorations of operations are reversed and input parameters become output parameters and vice versa.

In the profile the «base» / «conjugated» stereotype is defined for model elements of type Association. The following constraints apply: A port may only reference a protocol from a capsule, that is, it has exactly two association ends, one connected to a capsule and the other to a protocol. The association end connected to the capsule has aggregation type *aggregate* and the association is only navigable from the capsule to the protocol.

Communication Directions. In a protocol the «in» / «out» stereotypes described below are used to specify the ‘direction’ of operations. Although these stereotypes are present in the UML-RT draft(s), the semantics here is slightly different: we do not restrict the communication to simple, unidirectional signals, but preserve method call like communication of CSP-OZ with input and output parameters both possibly occurring in one method call (communication). So in our case these stereotypes specify, whether a communication is passive («in», waiting for a call) or active («out», initiating communication from a state machine) seen from the «base» side of a protocol.

In the profile the «in» / «out» stereotype is defined for model elements of type Operation. The following constraints apply: These operations must be abstract, that is, they have no associated implementation. This stereotype has the following tags: *inDecl* contains the declarations for the input parameters; *outDecl* contains the declarations for the output parameters; *addr* holds the declaration of the parameter used for addressing.

Communication Behaviour. For each operation in a protocol referenced by a capsule via «base» or «conjugated» ports, an equally stereotyped method has to exist in the capsule. It specifies the communication behaviour, i.e., the precondition for the communication to be allowed and the effect on the containing capsule’s state space.

In the profile the «base» / «conjugated» stereotype is defined for model elements of type Method. The following constraints apply: These methods must be owned by a capsule and may not be public. This stereotype has the following tags: *changes* lists the attributes of the capsule owning this method, which might be changed by the method; *enable* contains a Z predicate specifying the enable conditions for the method; *effect* contains a Z predicate describing the effect on the state space of the capsule owning this method.

Additional Conventions and Comments

In the following, additional features of the CSP-OZ adaption which are not part of the profile are discussed. First, in the statechart diagrams for the capsules the following conventions are used:

- Blocking semantics and synchronous communication. Communication between capsules (and therefore state machines) is synchronous and no events are silently discarded.
- Only the names of methods in the corresponding capsule may be used as triggers for transitions. Parameters are always omitted and there are no guards or actions.

The decision against guards and actions was made because it is closer to the CSP-OZ idea of separating data and control. It is of course still an option to include the contents of `enable` and `effect` tags of («`base`», «`conjugated`») methods as guards and actions in the state machine diagrams, but this would probably make the statecharts less readable.

2.2 UML Modelling for the Case Study

The case study “Holonc manufacturing system” is part of the German DFG priority program “Integration of specification techniques with applications in engineering”⁴. The system consists of two stores (*In* and *Out*), one for workpieces to be processed (the in-store), one for the finished workpieces (the out-store), a number of holonic transportation systems (hts) and machine tools for processing the workpieces. Every workpiece has to be processed by all the machine tools in a fixed order. The hts are responsible for transportation of workpieces between machine tools and stores. The hts work as autonomous agents, free to decide which machine tool to serve (within some chosen strategy). Initially, the in-store is full and the out-store as well as all machines are empty. When the in-store or a machine contains a (processed) workpiece it broadcasts to all hts a request to deliver this workpiece. The hts (when listening) send some offer to the machines, telling them their cost for satisfying the request. Upon receipt of such offers the machine decides for the best offer and gives this hts the order, which executes it and transports the workpiece to the next machine tool in the processing order. In this way, all workpieces are processed by all tools and transported from the in- to the out-store.

In this section the UML diagrams describing the transport element (‘Hts’) are presented. For a better readability, these diagrams do not show the internals (tags); for those details refer to section 3.⁵

The class diagram for the transport system is shown in Fig. 2. It contains all capsules used by the transport system and all protocols used by these capsules. Hts is a compound capsule containing instances of three other capsules (HtsCtrl, Driver, Acquisition). It has no attributes or methods of its own and even the ports are derived from the enclosed capsules, that is, messages are simply forwarded from and to the ports of the contained capsules.⁶ The internal architecture of Hts has to be specified in a structure diagram (Fig. 3). The three protocols

⁴ <http://tfs.cs.tu-berlin.de/projekte/indspec/SPP/index.html>

⁵ UML diagrams typically do not show larger textual items of a model (documentation, method bodies, etc.), since it would make the diagrams unreadable.

⁶ The outer ports are therefore called *relay* ports in ROOM and UML-RT.

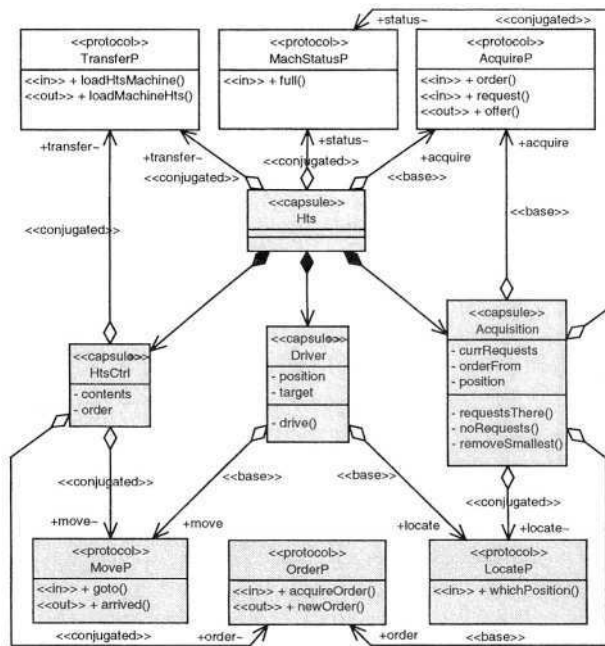


Fig. 2. Class Diagram: Transport

in the upper half of Fig. 2 are the main protocols of the whole system; they are used to communicate with the stores and machines and would show up in the class diagram(s) for the rest of the system as well. On the bottom of the diagram another three protocols are defined; they are used only for (Hts-) internal communication between the contained capsules. Thus Hts has no ports referencing these protocols.

The ROOM/UML-RT structure diagrams (e.g., Fig. 3) are used to specify the (communication) structure of instantiated systems or capsules. Capsules are rendered as boxes, with ports appearing as little boxes on their border. A port's colour visualises whether it refers to the «base» (black) or «conjugated» (white) protocol.

The state machines for the capsules specify the allowed communication sequences. For non-compound, concrete capsules a state machine is mandatory. Fig. 4 shows the state machine for HtsCtrl.

2.3 Translation to CSP-OZ

The translation to CSP-OZ encompasses three tasks: translating the state machines to CSP process expressions, generating CSP-OZ classes from the class diagrams (using the process expressions just mentioned), and translation of the structure diagrams.

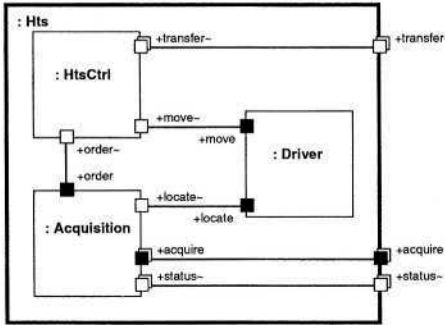


Fig. 3. Structure Diagram: Hts

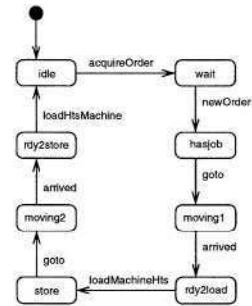


Fig. 4. Statechart: HtsCtrl

1. For simple protocol state machines (without concurrency and history, and using only completion transitions on compound states) a straightforward translation to CSP exists: For each state a process is defined as an external choice over all events occurring on transitions originating at the state, prefixed to the processes (represented by their name) for the target state. For the initial state on the top level of the statechart, the process `main` is created; final states correspond to the special CSP process `SKIP`, which represents termination. This can be extended to state machines with a restricted kind of concurrency (disjoint event sets in the concurrent submachines); the processes for the concurrent submachines are then put in parallel.

The general case needs a non-trivial translation; we have developed such a translation for a larger class of state machines [29]. Since the resulting CSP processes of this general translation scheme are much larger even for the simple state machines above, the simple translation is preferred.

2. Generation of the CSP-OZ classes consists mainly of assembling the tags attached to the various stereotyped model elements. Each operation of a protocol introduces a channel declaration in the CSP-OZ class using the protocol. There are three kinds of channels: `method` channels, which correspond to passive behaviour, i.e., communication will be initiated from the other end; `chan` channels, which correspond to invocation of a (remote) method; and `local_chan` channels, which are used for intra class communication. An operation marked as «in» in a protocol introduces a method channel declaration in the CSP-OZ class for a capsule with a «base» reference to this protocol and a corresponding channel channel declaration for a capsule with a «conjugated» reference (with the reversal of input and output parameters). With roles reversed, the same happens for operations marked «out». CSP-OZ inherit clauses are generated from the Generalization relationships. The CSP part already exists; composition of capsules is handled in step 3.
3. Translation of the structure diagrams is more complex; a general scheme is described in [13].

3 Specification

The CSP-OZ classes of the case study can thus be systematically generated from the UML model. Currently this has to be done by hand but an implementation of the profile within Rational Rose is under development. Next, we illustrate the translation by looking at the classes *HtsCtrl* and *Hts*. The translation first generates a given type *CRef* for every defined class *C*, containing *reference names* of instances, e.g., we have a type *HtsRef* for references to instances of *Hts* and types *MachineRef* and *ActiveMachineRef* for classes *Machine* and *ActiveMachine* which are in the full case study.

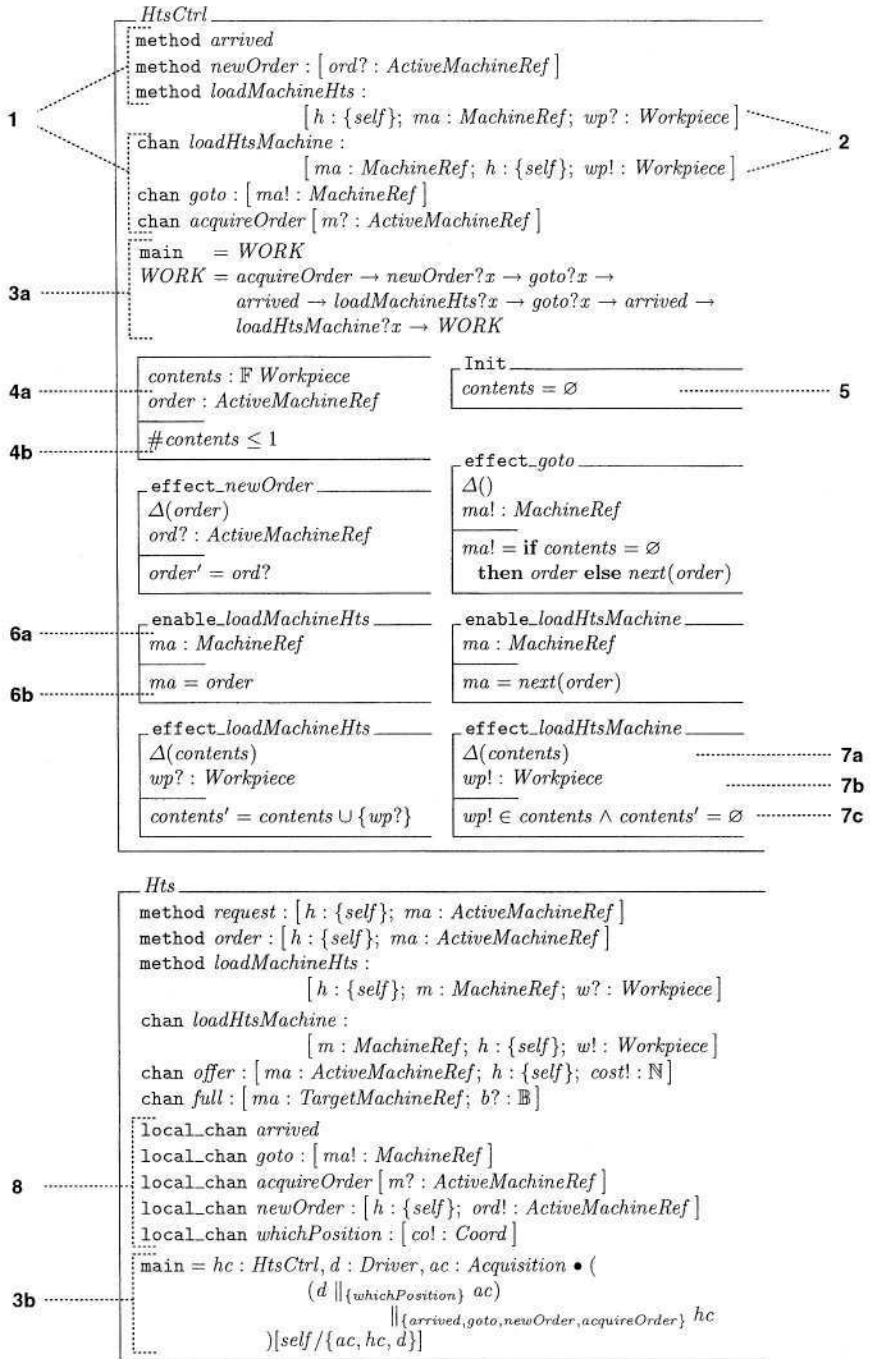
Next, for the capsules *HtsCtrl* and *Hts* in the UML model a CSP-OZ class is generated (Fig. 5). The first part of the specification of class *HtsCtrl* describes the basic interface according to the class diagram (Fig. 2). Then the attributes of the class are defined and the initial state values are given. The last part of class *HtsCtrl* specifies the enabling conditions (guards) and effects for the execution of methods. The variable *self* in the class specification for *HtsCtrl* (Fig. 5) is a special variable holding the instance name of objects of this class; for any instance of this class it can be regarded as a (unique) constant. Here, it is used for addressing objects: when a method is called on a channel and addressing is required (i.e., several objects can be reached via this channel), the first parameter will always be the instance name of the object.⁷ This is achieved by restricting the value of the first parameter to *self* in the capsule implementing the ‘method’ side of the operation. Another application of *self*, also used in the specification of *HtsCtrl*, is as a caller identification, enabling the callee to refer to the caller later.

To specify the class *Hts* we have to deal with instantiation of the three components *HtsCtrl*, *Driver* and *Acquisition* as depicted in Fig. 3. Since class *Hts* is a “pure” composition of *active* classes, it only has a process specification, but no data part. The interface is the union of the component interfaces, with all channels made local, which are not connected to the (relay) ports of *Hts*. The instantiation of the components is defined local to the main process. Thus the names of the component instances remain local to objects of class *Hts*. The final renaming [*self* / {*ac*, *hc*, *d*}] assures that all references to the instance names are replaced by a reference to the *Hts* object, effectively connecting the relay ports of *Hts* with the ports of the contained capsules.

If all tags in the profile have been properly filled in the UML model the CSP-OZ class specifications in Fig. 5 can be automatically generated. Using the numbers in Fig. 5 we look at the generation of the two classes in more detail:

1. Depending on «in» and «out» stereotypes in the protocol and on the «base» and «conjugated» stereotypes on the ports, a channel is declared as either method or chan.
2. The operation signature (type of data on the channel) is generated using the contents of the inDecl, outDecl and addr tags belonging to the corresponding protocol operation.

⁷ The declaration of this parameter is stored in the addr tag for operations on the UML level.

Fig. 5. CSP-OZ specifications for capsules *Hts* and *HtsCtrl*

3. For a simple capsule, the CSP part of the class contains the translation of the corresponding state machine (**a**); for compound capsules it contains the instantiation of the contained capsules (**b**).
4. The state schema is populated with the attributes of the capsule (**a**) and the class invariant is taken verbatim from the *invariant* tag of the «capsule» (**b**).
5. The init schema uses the predicate from the *init* tag of the capsule.
6. Enable schemas are generated using information from the corresponding protocol for the parameter declarations (**a**) and from the method's enable tag for the body (**b**).
7. Effect schemas are generated using the changes tag of the method for the Δ -list (**a**), information from the protocol for the parameter declarations (**b**) and from the method's effect tag for the body (**c**).
8. For internal connections in the structure diagram of a compound capsule local channel declarations are generated.

A larger part of the CSP-OZ specification of this case study can be found in [39]. Using a verification technique for CSP-OZ proposed in [14] the specification of the holonic manufacturing system has been verified, showing for instance deadlock freedom and adherence to the correct processing order.

4 Implementation

To link the CSP-OZ specification with a final Java implementation we now take a different approach. Instead of generating Java implementations from CSP-OZ we generate Java interfaces with *assertions*. The final (handwritten) implementation of these interfaces can then be monitored against the assertions. For writing the assertions we use two intermediate languages that support monitoring viz. runtime checking - JML and CSP_{jassda}.

The state-based part of CSP-OZ, formalised in Object-Z, is mapped to assertions written in the Java Modeling Language (JML) [24]. JML is a behavioural interface specification language (BISL), i.e., it provides “rich interfaces” that enrich the syntactic interface of a software module (signature) by a specification of its behaviour in the form of assertions (pre- and postconditions and invariants). These assertions can be checked at runtime [25]: whenever a violation of an assertion is detected an exception is thrown and the program terminates.

However, in JML the order of methods calls cannot be specified directly. To overcome this shortcoming, we complement JML by CSP_{jassda}, a generalisation of the trace assertion facility of Jass [2]. *Jassda* stands for *Java with assertions debugger architecture*. It uses the Java Debug Interface (JDI) to enable runtime checking by monitoring a program during execution and comparing the monitored behaviour with the CSP-like specification [5]. The CSP part of a CSP-OZ specification can thus be translated into trace assertions written in CSP_{jassda}. By combining the two runtime checking methods we guarantee that the current program run performs correct data modification (JML) in the correct order (CSP_{jassda}) – until we detect a violation of the specification (s).

Both formalisms have in common that the specification of the Java program is separated from its implementation. This allows one to switch to an alternative implementation while keeping the same specification.

4.1 From Object-Z to JML

JML annotates normal Java programs with special types of comments so that the same annotated programs may be used both for compilation by an ordinary Java compiler and for the JML tools. Besides method pre- and postconditions and class invariants, JML also provides *model variables*. These variables are accessible in the specification only and describe the *abstract* state of instances of a type. In contrast to normal Java class attributes these model variables can also be used in interfaces and thus describe (part of) the abstract state of an implementing Java class.

This kind of abstraction, hiding the concrete implementation of the state space, fits very well with Object-Z. It enables us to use an almost direct mapping from the abstract Object-Z state space to the abstract state space of a JML interface, that is finally mapped to the concrete state space of the implementation by an abstraction relation, i.e., a JML *represents clause*.

The translation of CSP-OZ to JML defines a Java interface with JML specification for every CSP-OZ class. These interfaces have to be implemented by the final Java program.

We do not treat the types of reference names in the translation directly, but instead we identify them with the Java types of the classes they are referring to. Whenever we define a Java type class *C* or interface *C*, we define a reference to that type with name *ref* by the declaration *C ref* ; .

In the following we use the CSP-OZ class *HtsCtrl* of Fig. 5 as the running example to explain the translation to a JML/Java interface *HtsCtrlSpec*. The interface definition of the CSP-OZ class is translated into the interface of a Java class. Since we want to be able to associate a specification with all possible communications we have to translate both types, method and chan, to Java methods. The schema type of a channel is translated into the formal parameters and the return value of the method. References to *self* are omitted because a Java class “knows” itself (*this*), other reference parameters become formal parameters of the method. The same holds for the input parameter. The output parameters are translated into the return values of the Java methods. Since Java methods may only have exactly one return value, we have to translate more than one output parameter to a Cartesian product of participating outputs. This is summarised by the following rule.

Translation Rule 1 *A CSP-OZ class or Object-Z class C is translated to a JML interface CSpec extending either JMLType or the specification classes from the inherit-clause. Every method or channel m of C is represented by a Java method. Let [ref : {self}; other : OtherRef; in? : In; out? : Out] be the schema type of m with InType, OutType as JML types for In and Out, then the Java method will get the signature Out Type m (OtherSpec other, InType in) .*

Translation of the schemas starts with the state schema that will be transformed into model variables of our specification. In our running example we use JML's `JMLValueSet` for sets of arbitrary objects to model the given type *Workpiece*. Since Java does not provide generic types, like C++ templates, we cannot reflect the type of *contents*' elements more precisely⁸. The predicate part of the schema and the implicit predicates that we get by normalisation of the schema are translated into class invariants. Finally, the init schema is translated into initial conditions for the model variables.

Translation Rule 2 *The declarations within the state schema of a CSP-OZ class or Object-Z class become model variables in the JML specification of the class. The predicates of the normalised state schema become class invariants of the JML type. The predicates of the normalised Init schema are translated into initially clauses of the model variables.*

To complete the “rich” interface `HtsCtrlSpec` we add the method specifications. For each operation the enable schema yields the precondition of the method indicated by the keyword `requires`. The effect schema is translated into the postcondition indicated by the keyword `ensures`, where all references to the pre-state (the unprimed variables) have to be enclosed by the operator `\old()`.

Translation Rule 3 *For every channel or method m we translate the `enable_m` and `effect_m` schemas to precondition, postcondition and assignable clause of the Java method m . For the precondition we have to translate `enable_m` to a Java boolean expression. For the postcondition we translate `effect_m` to a Java boolean expression where references to the pre-state are enclosed by `\old()`. The assignable clause is just the Δ -list.*

Fig. 6 shows (part of) the JML specification that the translation rules produce for the CSP-OZ class *HtsCtrl*. In some cases we cannot translate the specification directly. In particular, set comprehensions pose a problem because they imply a quantification over the set's elements. In that case we have to find an equivalent specification that makes the quantification explicit and thus can be translated to JML expressions. In the case study there was only one problematic expression for which it was simple to find an equivalent translatable expression.

4.2 Linking JML Specification and Implementation

To link an actual implementation with our JML specification two things need to be done. We must implement the interface(s) of our JML specification and thus provide an implementation for the methods specified there. Second, we have to fill the specification with life by giving the relation between model variables occurring in the JML specification and attributes of the implementing classes. The implementation should be a *data refinement* of the JML specification. At

⁸ A generic type concept (http://java.sun.com/aboutJava/communityprocess/jsr/jsr_014_gener.html) will be part of the upcoming J2SE 1.5 Release. Also some investigations of the formal aspects of such a facility were done.

```

package hms.spec;
/*@ model import org.jmlspecs.models.JMLValueSet;
/*@ model import org.jmlspecs.models.JMLType;

public interface HtsCtrlSpec /*@ extends JMLType @*/{

    /*@ public model instance JMLValueSet contents
    /*@ public initially contents.isEmpty();
    /*@ public model instance MachineSpec order;

    /*@ public invariant contents.size() <= 1;
    ...
    /*@ public normal_behavior
        requires ma.equals(order);
        assignable contents;
        ensures contents.equals(\old(contents.union(new JMLValueSet(wp)))); @*/
    public void loadMachineHts(MachineSpec ma, WorkpieceSpec wp);

    /*@ public normal_behavior
        requires ma.equals(order.next());
        assignable contents;
        ensures \old(contents).has(\result) && contents.isEmpty(); @*/
    public WorkpieceSpec loadHtsMachine(MachineSpec ma);
    ...
}

```

Fig. 6. JML specification `HtsCtrlSpec` for class `HtsCtrl`

runtime it is checked that the concrete implementation adheres to the abstract specification via a representation relation. The JML represents clause links model variables with the implementation and thus defines this representation relation.

This clause usually is a function mapping attributes of the implementing class to the model variables by giving an expression that returns the required type. It is also possible to use a representation *relation*, for which JML provides the keyword `\such_that`. Proper representation relations are, however, hard to handle in automatic program verification as shown in [3] and for the same reasons in a runtime assertion checker and thus should be avoided.

Below, the representation relation between a class `HtsCtrl` and the JML specification `HtsCtrlSpec` is given. The class defines a concrete attribute `cont` that is used to represent the model field `contents`. The expression `(cont == null ? new JMLValueSet() : new JMLValueSet(cont))` maps the concrete attribute to an appropriate type for the model field.

```

package hms.impl;
import hms.spec.*;
/*@ model import org.jmlspecs.models.JMLValueSet;

public class HtsCtrl extends Thread implements HtsCtrlSpec {
    protected WorkpieceSpec cont;
    /*@ protected instance represents contents <-
        (cont == null ? new JMLValueSet() : new JMLValueSet(cont)); @*/

```

4.3 From CSP to CSP_{jassda}

In this section we will show how to link the specification with the dynamic behaviour of the implementation. In CSP-OZ we use CSP processes to specify

the order of communications between active objects, thus giving an event based view on the specification.

In the JML part we mapped communication channels to methods of Java interfaces. To observe communication of our objects we have to observe invocations of these methods. More precisely, the events to observe are start and end of a method invocation (a method is pushed to the method stack or popped). A communication from a sender to a receiver will result in four events: the communication starts at the caller's side before it starts at the receiver's side, and both calls end before a new one begins.

To specify such an order on method start and end events we developed CSP_{jassda} [27]. This language is, like JML, somewhat specific to Java⁹ and very close to CSP so that it will be easy to translate the specification. The design of CSP_{jassda} is closely coupled with the development of the tool *jassda* [4,5]. The *jassda* tool is able to test whether a program generates a given set of events in the correct order according to the specification given in CSP_{jassda} . This test is performed at the byte-code level so that even for a different set of events no modification or instrumentation of the code is required.

In every class of the specification the CSP process *main* describes the possible communication behaviour of that class and thus the order of method invocations at the JML/Java level. For the translation we have to distinguish two types of processes that we get from our translation: *simple class processes* and *instantiation processes*. A simple class process describes the behaviour of one instance of a class without any assumption about its environment. Instantiation processes occur for every instantiation of subcomponents of a class, i.e., they result from the translation of structure diagrams.

Simple class processes. A simple class process just describes the order of events that are produced by exactly one instance of exactly that class. Thus we need to allow this main process to run for every instance. This is done by a parallel composition of a parameterised process that is instantiated for each instance of a class that emits a relevant event. The parameter is used to restrict the alphabet of that subprocess to the events of one instance of the class.

In a simple class process only methods of the class are mentioned and calls to these methods must end before we call another method. Therefore we abbreviate the CSP_{jassda} specification with one that has the same structure as the CSP-OZ process. To use the *jassda* trace checker we expand the abbreviated specification to the full CSP_{jassda} specification by a preprocessor. For example, the simple class process of *HtsCtrl* (cf. Fig. 5) is translated into the following abbreviated CSP_{jassda} specification.

```
public interface HtsCtrlSpec {
    public void acquireOrder();
    public void newOrder(ActiveMachineSpec newOrder);
    public MachineSpec gotoMachine();
    public void arrived();
}
```

⁹ Although it will be possible to transfer most of the concepts to another programming language.

```

public void loadMachineHts(MachineSpec ma, WorkpieceSpec wp);
public WorkpieceSpec loadHtsMachine(MachineSpec ma);
/*% public csp
  main() { WORK() }
  WORK() { acquireOrder -> newOrder -> gotoMachine ->
    arrived -> loadMachineHts -> gotoMachine -> arrived ->
    loadHtsMachine -> WORK() }    %*/
}

```

Instantiation processes – synchronisation and delegation. To translate a class that instantiates other classes, i.e., processes, we have to take the interfaces of the instantiated processes into account. In our example *Hts* instantiates *HtsCtrl* with reference name *hc*, and *Driver* with reference name *d* and *Acquisition* with reference name *ac* (cf. Fig. 5). The connection diagram of this CSP-OZ process corresponds to the structure diagram of class *Hts* (Fig. 3). The Java methods represent the ports from the diagram. A communication event on a channel is represented by four events of the corresponding Java methods (start and end of the methods on both sides of the channel). Thus a CSP channel is specified by an order on these events. So we specify a recursive processes for each channel and compose them in parallel.

For every such instantiation process we have three types of channels:

- For a channel that is local to one of the instances there is no connection to model and thus it is not considered.
- For every channel *m* that instances *a* of a CSP-OZ class *A* and *b* of a CSP-OZ class *B* synchronise on we need to link the Java methods *m* of classes *ASpec* and *BSpec*.

The synchronisation is represented by subsequent calls to the methods of the two classes where we pass the result of the first call of the method (the output of the method side) as argument to the call of the second method (the input of the chan side). The process accepting this kind of event traces is abbreviated to *SYNC(a,b,m)*.

In our example the instances *hc* (class *HtsCtrl*) and *d* (class *Driver*) synchronise on channel *arrived*, so we add the process *SYNC(hc,d,arrived)* to the specification of *Hts*.

- If CSP-OZ class *C* instantiates a class *A* with reference name *a* and a channel *m* of *A* is visible to the outside world it has to be “exported” by the Java class *CSpec*. In this case *CSpec* must forward the call to the instance *a* (that provides the method). Forwarding means that *CSpec* provides a method *m* with the same signature as method *m* of *ASpec*, every call to *CSpec.m* is followed by a call to *a.m* and the latter call ends before the first one (abbreviation: *EXPORT(a,m)*).

For the class *Hts* of the case study this means that channel *offer* has to be exported from instance *ac* and therefore we add an *EXPORT(ac,offer)* process to the specification of *Hts*.

Using this translation we obtain a *CSP_{jassda}* process for every class describing the communication behaviour of every instance of this class. To obtain the specification of the whole system we let all these processes run in parallel.

4.4 Execution with Runtime Checks

Fig. 7 summarises the process from CSP-OZ to Java executables with runtime checks using JML and jassda. We translate the CSP-OZ specification into a CSP_{jassda} and a JML specification, i.e., Java interfaces annotated with assertions. Then we have to provide Java code that implements¹⁰ the (Java part of the) JML specification. We translate both to Java byte code using the JML runtime assertion compiler and get a byte code program that performs checks for the data part at runtime. Then we run the program while it is connected to the jassda tool that checks the dynamic behaviour against the CSP_{jassda} specification.

This approach lets the user run the program as if executed without the runtime checks and thus requires inputs to be given by the user. For reactive systems, like the one from the case study, there is no external input and thus this is no limitation. Utilising the `jml-unit` tool the generated JML specifications would also support unit tests. In that case the expected output would be derived from the specification. Generating appropriate test patterns is still needed in this case. Generating test patterns could benefit from the formal CSP-OZ model, but this is another subject of research.

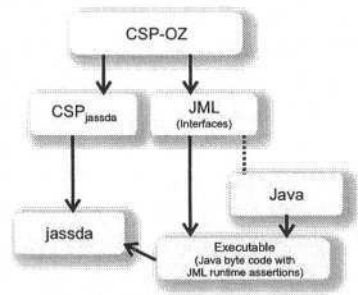


Fig. 7. From CSP-OZ to Java with runtime checks

5 Conclusion

In this paper we described part of the modelling, specification and implementation of the case study “Holonc Manufacturing System”. The case study served as an illustration for our approach of integrating a formal method, here CSP-OZ, into a UML and Java based software development. It demonstrated how UML’s multi-view modelling facilities (static *and* dynamic behaviour) can be adequately reflected in a formal specification (using an *integrated* formal method) as well as monitored in the final implementation (using one tool for runtime checking of static and another for dynamic behaviour).

Tool support. As a platform for integration of the CSP-OZ profile presented here we have chosen the UML tool *Rational Rose*. Its *extensibility interface* allows the portable customisation of the user interface, implementing new menus and dialogs, and the generation of CSP-OZ code directly from the editor. A first prototype of a CSP-OZ tool in Rose exists, but it does not yet support the full profile. The idea of extending a UML tool with facilities for editing Z or Object-Z specifications appeared already in [10] but this work covered neither state machines nor structure diagrams. Another work in this direction is [37] which translates UML diagrams (annotated with B) into B.

¹⁰ This means implements in the sense of Java’s `implements`.

Related work. Developing formal semantics for UML is currently a very actively pursued research topic (see for instance [23,34,31]). Especially CSP is a prominent choice as a semantic model for UML diagrams [7,11]. Here, we differ in that our choice for CSP-OZ enables a usage of different, orthogonal diagram types of UML: CSP-OZ can be used to give a semantics to class diagrams (modelling static behaviour) in combination with state machines (dynamic behaviour) and structure diagrams (modelling the architecture). The basic ingredients of UML needed for modelling reactive systems are thus covered. The translation to CSP-OZ furthermore opens the possibility of formally checking consistency between the different diagrams [30].

In the context of Java, formal approaches are most often applied in order to verify Java programs. These approaches either use a Hoare-logic for specifying properties and develop proof support with theorem provers [28,20] or they apply model checking techniques to Java programs [17]. An approach combining theorem proving with full automation is the static checker ESC/Java [26].

Besides JML a number of other languages and tools exist which provide “Design-by-contract”-extensions for Java (for instance [22]). JML is, however, the most widely accepted language, and its concept of model variables proves to be an ideal tool for bridging the gap between formal specification and programming language. For runtime verification of the *dynamic* behaviour of Java programs there are tools that monitor programs against temporal logic formulae [8,18]. In the context of UML which specifies dynamic behaviour using state machines (or other interaction diagrams) runtime monitoring against CSP expressions as done by jassda seems more adequate.

A different approach for combining CSP-OZ with Java is taken by [6] which transform CSP-OZ specifications into CTJ, an extension of Java with CSP-like processes and channels, using a number of refinement rules.

However, none of the works mentioned above combine all three levels in one approach. Our primary aim is to smoothly integrate the formal method into a software development process with UML *and* Java. Both the modelling and the implementation should benefit from the formal specification, achieving a higher degree of correctness in the resulting design and software.

Acknowledgements. Thanks to John Knudsen for helpful comments.

References

1. E. Ábrahám-Mumm, F.S. de Boer, W.-P. de Roever, and M. Steffen. Verification for Java’s reentrant multithreading concept. In *FoSSACS 2002*, volume 2303 of *LNCS*, pages 4–20. Springer, 2002.
2. D. Bartetzko, C. Fischer, M. Möller, and H. Wehrheim. Jass – Java with Assertions. In Klaus Havelund and Grigore Roşu, editors, *ENTCS*, volume 55. Elsevier, 2001. <http://www.elsevier.nl/locate/entcs/volume55.html>.
3. C.-B. Breunesse and E. Poll. Verifying JML specifications with model fields. In *Workshop on Formal Techniques for Java-like Programs – FTfJP’2003*. ETH Zürich, July 2003. Technical Report 108.

4. M. Brörkens. Trace- und Zeit-Zusicherungen beim Programmieren mit Vertrag. Master's thesis, Univ. of Oldenburg, Dept. of Computing Science, January 2002.
5. M. Brörkens and M. Möller. Dynamic Event Generation for Runtime Checking using the JDI. In Klaus Havelund and Grigore Rosu, editors, *ENTCS*, volume 70. Elsevier, 2002. <http://www.elsevier.nl/locate/entcs/volume70.html>.
6. A. Cavalcanti and A. Sampaio. From CSP-OZ to Java with Processes. In *Workshop on Formal Methods for Parallel Programming, held in conjunction with International Parallel and Distributed Processing Symposium*. IEEE CS Press, 2002. Contained in IPDPS collects proceedings CD-ROM.
7. J. Davies and Ch. Crichton. Concurrency and Refinement in the Unified Modeling Language. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *ENTCS*, volume 70. Elsevier, 2002.
8. D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Modelchecking and Software Verification*, number 1885 in LNCS, pages 323–330. Springer, 2000.
9. R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. *Computer Standards and Interfaces*, 17:511–533, 1995.
10. S. Dupuy, Y. Ledru, and M. Chabre-Peccoud. An overview of RoZ - a tool for integrating UML and Z specifications. In *12th Conference on Advanced information Systems Engineering (CAiSE'2000)*, 2000.
11. G. Engels, J. Küster, R. Heckel, and L. Groenewegen. A Methodology for Specifying and Analyzing Consistency of Object-Oriented Behavioral Models. In *9th ACM SigSoft Symposium on Foundations of Software Engineering*, volume 26 of *ACM Software Engineering Notes*, 2001.
12. C. Fischer. CSP-OZ: A combination of Object-Z and CSP. In H. Bowman and J. Derrick, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS '97)*, volume 2, pages 423–438. Chapman & Hall, 1997.
13. C. Fischer, E.-R. Olderog, and H. Wehrheim. A CSP view on UML-RT structure diagrams. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE'01)*, volume 2029 of LNCS, pages 91–108. Springer, 2001.
14. C. Fischer and H. Wehrheim. Model-checking CSP-OZ specifications with FDR. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proc. 1st International Conference on Integrated Formal Methods (IFM)*, pages 315–334. Springer, 1999.
15. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, second edition, June 2000.
16. G. Gullekson. Designing for concurrency and distribution with Rational Rose RealTime. Technical report, Rational Software, 2000.
17. J. Hatcliff and M. Dwyer. Using the Bandera tool set to model-check properties of concurrent Java software. In K.G. Larsen, editor, *CONCUR 2001*, LNCS. Springer, 2001.
18. K. Havelund and G. Rosu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Rosu, editors, *ENTCS*, volume 55. Elsevier, 2001.
19. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
20. M. Huisman and B. Jacobs. Java Program Verification via a Hoare Logic with Abrupt Termination. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of LNCS, pages 284–303. Springer, 2000.
21. B. Jacobs, J. van den Berg, M. Huisman Martijn van Berkum, U. Hensel, and H. Tews. Reasoning about Java classes (preliminary report). In *Proc. OOPSLA 98*, volume 33 of *ACM SIGPLAN Notices*, pages 329–340, Oct. 1998.
22. R. Kramer. iContract - the Java Design by Contract tool. Technical report, Reliable Systems, 1998.

23. D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11:430–445, 1999.
24. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. Technical Report 98-06v, Iowa State Univ., Dept. of Computer Science, May 2003. See <http://www.jmlspecs.org>.
25. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accomodates both runtime assertion checking and formal verification. In *FMCO'02*, LNCS, 2003. To appear.
26. K. R. M. Leino. Extended static checking: A ten-year perspective. In Reinhard Wilhelm, editor, *Informatics – 10 Years Back, 10 Years Ahead*, volume 2000 of LNCS, pages 157–175. Springer, 2001.
27. M. Möller. Specifying and Checking Java using CSP. In *Workshop on Formal Techniques for Java-like Programs – FTfJP'2002*. Computing Science Department, University of Nijmegen, June 2002. Technical Report NIII-R0204.
28. A. Poetzsch-Heffter and J. Meyer. Interactive verification environments for object-oriented languages. *Journal of Universal Computer Science*, 5(3):208–225, 1999.
29. H. Rasch. Translating UML state machines into CSP. Technical report, Universität Oldenburg, 2003.
30. H. Rasch and H. Wehrheim. Checking Consistency in UML Diagrams: Classes and State Machines. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems (FMOODS'03)*, volume 2884 of LNCS, pages 229–243. Springer, 2003.
31. G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE 2000)*, volume 1783 of LNCS. Springer, 2000.
32. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
33. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison-Wesley, 1999.
34. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. In S.D. Stoller and W. Visser, editors, *ENTCS*, volume 55. Elsevier, 2001.
35. B. Selic, G. Gullekson, and P. T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
36. B. Selic and J. Rumbaugh. Using UML for modeling complex real-time systems. Technical report, ObjecTime, 1998.
37. C. Snook and M. Butler. Tool-Supported Use of UML for Constructing B Specifications. Technical report. <http://www.ecs.soton.ac.uk/~mjb/U2Bpaper2.pdf>.
38. OMG Unified Modeling Language specification, version 1.5, March 2003. <http://www.omg.org>.
39. H. Wehrheim. Specification of an automatic manufacturing system – a case study in using integrated formal methods. In T. Maibaum, editor, *Fundamental Approaches of Software Engineering (FASE 2000)*, volume 1783 of LNCS, pages 334–348. Springer, 2000.

Object-Oriented Modelling with High-Level Modular Petri Nets

Cécile Bui Thanh¹ and Hanna Klaudel²

¹ LACL, Université Paris 12
61, avenue du général de Gaulle – 94010 Créteil, France
bui@univ-paris12.fr

² LaMI, Université Evry-Val d'Essonne
523, Place des Terrasses – 91000 Evry, France
klaudel@lami.univ-evry.fr

Abstract. In this paper, we address the problem of expressing object-oriented concepts in terms of Petri nets. This is interesting, first, as a possibility of representing concurrent system specifications written in object-oriented formalisms or languages with Petri nets, and second, as a way of allowing automated verification of the obtained Petri net using existing reachability analysis tools.

We start from an existing parallel specification language having a modular Petri net semantics and we extend it with object-oriented features inspired from Java and C++. The translation of these new extensions into the Petri net domain is given using a class of modular coloured Petri nets and includes, in particular, a treatment of inheritance and of dynamic binding.

Keywords. Object-orientation, coloured Petri nets, semantics.

1 Introduction

In this paper, we propose a way to express object-oriented concepts, like inheritance and dynamic bindings, in terms of Petri nets. The motivation is to provide a translation of concurrent system specifications written in object-oriented formalisms or languages into Petri nets, and thus, to allow automated verification of the obtained Petri net using existing reachability analysis tools [8,19,16].

The starting point of our approach is the high-level parallel specification language $B(PN)^2$ [1,10] (*Basic Petri Net Programming Notation*). It comprises most traditional concepts of parallel programming, like parallel composition, iteration, guarded commands, procedures and communications. Thanks to its simplicity, it can easily be used as a basis for various extensions and the results found for it may then be applied to “real-life” languages. Another advantage of $B(PN)^2$ is that it has already a concurrent formal semantics in terms of a class of high-level (coloured) Petri nets, called *M-nets* [2]. The particularity of M-nets is that they are provided with a set of composition operations and allow to represent large (possibly infinite) systems in a compact and structured

way. Moreover, $B(PN)^2$ and M-nets are implemented in the PEP toolkit [8], allowing to simulate a modelled system and also to verify its properties *via* model checking.

In this paper we propose an extension of $B(PN)^2$, called the Basic Object-Oriented Notation (BOON), having a syntax inspired from Java [9] and C++[18], and a semantics in terms of M-nets. This extension allows for defining classes with their own fields (attributes and methods), single class inheritance, polymorphism, and dynamic binding. The proposed semantics is modular, in particular, each class is represented by a module, itself composed of various submodules, each one representing either an attribute or a method of the class, or a mechanism intended for handling inheritance or management of instances (objects). All these modules are combined thanks to the powerful M-net synchronisation mechanism leading to a (large but structured) coloured Petri net. It may be seen as an alternative to other Petri net based formalisms capable to express object-oriented concepts, which often use more complex net classes. This is the case, for instance, for *Object Petri Nets* (OPN) [11], whose nets are enriched with net tokens, or for CO-OPN [3] and CLOWN [6], which use algebraic Petri nets (nets extended with algebraic data types).

The paper provides also a discussion concerning the soundness of this extension, in particular, concerning the correctness of the handling of dynamic bindings. In this respect, it perceptibly improves the first attempts in defining an object-oriented version of $B(PN)^2$ proposed in [14,13].

2 Syntax and Semantics of $B(PN)^2$

$B(PN)^2$ is a parallel programming language comprising shared memory parallelism, channel communication and allowing the nesting of parallel operators, blocks and procedures. The following is a fragment of the syntax of $B(PN)^2$ (with keywords typeset in **bold** face, non-terminals in roman face and *italic* denoting values supplied by the program):

```

program ::= program block
block    ::= begin scope end
scope    ::= com | decl ; scope
com      ::= <expr> | P(arglist) | block | (com) | do alt-set od
           | com || com | com ; com

```

An atomic command is a $B(PN)^2$ expression “<expr>”, *i. e.*, a term constructed over *operators*, *constants* (from a given set $\mathbb{V}$) and *program variables*, which can be executed if the expression evaluates to true. A program variable¹ v can appear in an expression as $'v$ (pre-value) or v' (post-value), denoting respectively its value just before and just after performing the command during the program execution. It may also appear just as v if the command does not change its value

¹ Originally, $B(PN)^2$ supports also channel variables, which are omitted here because they may be treated in a different manner in an object-oriented environment.

(if it is just read). Thus, for example, $\langle 'v > 0 \wedge v' = w \rangle$ corresponds to an atomic statement which requires the variable v to be greater than zero in which case the value of the variable w is assigned to v .

A command “com” is either an atomic command, a procedure call, one of a number of command compositions, or a block comprising some declarations for a command. Parentheses allow to combine arbitrarily the various command compositions.

The domain of relevance of a variable or a procedure identifier is limited to the part of a $B(PN)^2$ program, called “scope”, which follows its declaration. As usual, a declaration, in a new block, with an already used identifier results in the masking of the existing identifier by the new one. A declaration of a program variable v is made with “**var** $v : V$ ”, where $V \subseteq \mathbb{V}$ is a set of values, while that of a procedure P is made with “**proc** $P(\text{parlist})$ block”, where “parlist” is the list of formal parameters of P .

Besides traditional control flow constructs, like sequence and parallel composition, there is a command “**do...od**” which allows to express all types of loops and conditional statements. The core of this statement is a set of clauses of two types: repeat commands, “com; **repeat**”, and exit commands, “com; **exit**”. During an execution, there can be zero or more iterations, each of them being an execution of one of the repeat commands. The loop is terminated by an execution of one of the exit commands. Each repeat and exit command is typically a sequence with an initial atomic action, the executability of which determining whether that repeat or exit command can start. If several are possible, there is a non-deterministic choice between them.

2.1 Existing M-net Based Semantics of $B(PN)^2$

M-nets [2] form a class of high-level (coloured) Petri nets provided with a set of operations giving to them an algebraic structure. Like other high-level Petri net models, M-nets carry the usual annotations on places (sets of allowed tokens), arcs (multisets² of annotations) and transitions (guards³). In addition, places have a status (entry, exit or internal) used for net compositions; transitions carry labels used for inter-process communications, which are similar to CCS ones [17] but extended to (multi)sets of actions with arbitrary arity.

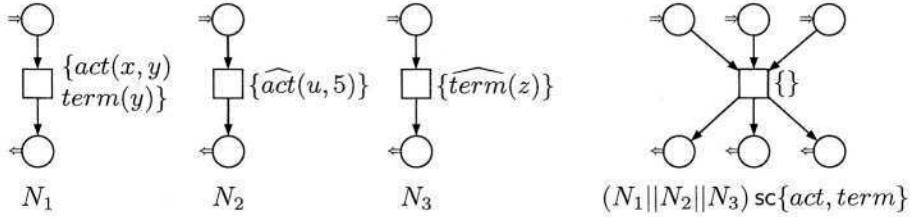
The communications can be enforced using the operation of *scoping* w.r.t. a set of actions, which intuitively corresponds to a set of binary *synchronisations* involving matching⁴ pairs of actions, *e.g.*, $act(x, y)$ and $\widehat{act}(u, 5)$, followed by the *restrictions*. For instance, the synchronisation w.r.t. $\{act\}$ applied to a net containing the transitions with labels $\{act(x, y), term(y)\}$ and $\{\widehat{act}(u, 5)\}$ will

² A multiset is formally a function m which gives to each element of a set E the number of its occurrences. We will use for multisets the extended set notation, *e.g.*, $\{a, a, b\}$ for $m(a) = 2$, $m(b) = 1$ and $m(x) = 0$ for all $x \in E \setminus \{a, b\}$.

³ A guard is a Boolean expression which plays the role of an occurrence condition.

⁴ Actions are “matching” if their parameters can be componentwise unified, *e.g.*, actions $act(x, 6)$ and $\widehat{act}(u, 5)$ are not matching and cannot synchronise.

produce in the net a new transition labelled $\{term(5)\}$, obtained by gluing the two former transitions together, while the restriction w.r.t. $\{act\}$ will remove from the resulting net the transitions whose labels involve act or $\widehat{act}$. The same operations but w.r.t. $\{act, term\}$ applied to a net containing the transitions with labels $\{act(x, y), term(y)\}$, $\{\widehat{act}(u, 5)\}$ and $\{\widehat{term}(z)\}$ will produce in this net a new transition (with empty label), corresponding to a three-way synchronisation, and remove from the net all transitions whose labels involve act or $term$, see below.



The marking of an M-net associates to each place a multiset of values (tokens) from the type of the place and the transition rule is like for other high-level nets. A transition t can be executed if the inscriptions of its input arcs evaluate to values which are present in the input places of t and if the guard of t evaluates to true. The execution of t transforms the marking by removing values (accordingly to the evaluation of arc inscriptions) from the input places of t and by depositing values in its output places.

The M-net semantics of a $B(PN)^2$ program is defined compositionally in [1] through the semantical function $Mnet$. The main idea in describing a block is (i) to juxtapose the nets for its local resource declarations (variables and procedures) with the net for its command followed by a termination net for the declared variables and procedures, (ii) to synchronize all matching data/command transitions and to restrict these transitions in order to make local variables invisible outside the block and (iii) to add the initial marking to the obtained net (typically a black token $\bullet$ in each initial place).

Each variable or procedure declaration is translated into a corresponding *resource M-net*. For instance, the declaration of a variable v of the type $V \subseteq \mathbb{V}$ gives rise to the M-net $N_{v,V}$ represented in Fig. 1. The current value of the variable v is stored in the central place of type V and may be updated using the $\{\widehat{access.v}(x, y)\}$ -labelled transition. The action $\widehat{access.v}(x, y)$ describes the change of value of v from its current value x , to the new value y .

The declaration procedure P is translated into a procedure resource M-net N_P , composed itself of resource M-nets for all local variables, of the M-net representing the body (command) of the procedure and of the M-net managing various procedure instances. A new procedure instance is started in N_P thanks to the action $\widehat{call\_P}(pid, f_1, \dots, f_n)$, where pid denotes the procedure instance identifier provided by N_P and $f_1, \dots, f_n$ represents the list of net variables corresponding to the formal parameters intended to be substituted with the effective ones. A call to P is translated into a call M-net $N_{call,P}$, which triggers a new instance of the procedure through the action $\widehat{call\_P}(pid, e_1, \dots, e_n)$, where $e_1, \dots, e_n$ is

the list of values or net variables corresponding to the effective arguments of P . The synchronisation between $\widehat{call\_P}(pid, f_1, \dots, f_n)$ and $call\_P(pid, e_1, \dots, e_n)$ substitutes the formal parameters with effective ones, which ensures a correct initialisation of each procedure instance.

Knowing this will be enough for our purpose; more details concerning the semantics of procedures can be found in [10,7].

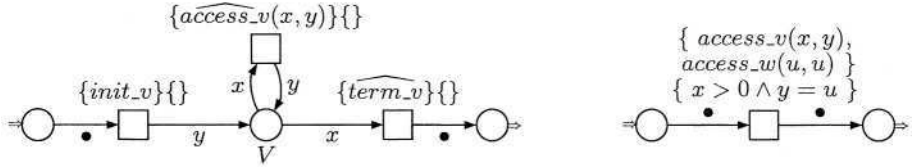


Fig. 1. The resource M-net $N_{v,V}$ of the variable v of the type V and the M-net $N_{\{access\_v(x,y), access\_w(u,u)\}\{x>0\wedge y=u\}}$ of the atomic command $\langle'v > 0 \wedge v' = w\rangle$

Sequential and parallel compositions are directly translated into the corresponding net operations, *e.g.*, $Mnet(com_1; com_2) = Mnet(com_1); Mnet(com_2)$, while the semantics of the “do ... od” construct involves the M-net iteration operator, which allows to express various kinds of loops, and the choice operator. The semantics of an atomic command “ $\langle expr \rangle$ ” is the M-net $N_{\alpha\gamma}$ where α is a set of actions corresponding to the access to the program variables involved in “ $expr$ ”, and γ is the guard obtained from “ $expr$ ” with program variables appropriately replaced by net variables, like *e.g.*, x for v , y for v' and u for w in:

$$Mnet(\langle'v > 0 \wedge v' = w\rangle) = N_{\{access\_v(x,y), access\_w(u,u)\}\{x>0\wedge y=u\}}.$$

The M-net above has one transition as shown in figure 1. Its label is used for a communication with the resource nets for variables v and w : x is read and y is written with the action $access\_v(x,y)$ (because v is updated) and u is read and written with the action $access\_w(u,u)$ (because w is unchanged). The guard ensures that $x > 0$ and that y is set to u .

For instance, the essential part⁵ of the M-net semantics of

```

program
begin
  var  $v, w : V$ 
   $((v' = 3) || (w' = 0)) ; \langle'v > 0 \wedge v' = w\rangle$ 
end

```

is the initially marked (with one token $\bullet$ in each entry place) M-net:

$$(N_{v,V} \parallel N_{w,V} \parallel ((N_{\{access\_v(x,y)\}\{y=3\}} \parallel N_{\{access\_w(x,y)\}\{y=0\}}) ; N_{\{access\_v(x,y), access\_w(u,u)\}\{x>0\wedge y=u\}})) \text{sc}\{access\_v, access\_w\},$$

where $\text{sc}\{access\_v, access\_w\}$ is the scoping w.r.t $\{access\_v, access\_w\}$.

⁵ The complete semantics takes into account the initialisation and the termination of the variables, which are omitted here.

3 Object-Oriented Extension

In order to introduce our object-oriented extension, we fix first a syntax at the $B(PN)^2$ level. Next, we provide this syntax with a high-level Petri net semantics using M-net algebra. The new notation, inspired from Java and C++, will be called Basic Object-Oriented Notation (BOON).

3.1 Syntax of the Object-Oriented Features

The main extension we propose concerns the introduction in $B(PN)^2$ of concepts of *classes*, *objects*, *inheritance*, *polymorphism* and *dynamic binding*.

A class is a high-level abstraction defined by a set of characteristics and services, called *fields*. The characteristic fields of a class are called the *attributes*, and the service ones are called the *methods*. We assume that each class has a *name* in a set $\mathbb{C} \subseteq \mathbb{V}$ and will use the letters $C, D, \dots$ for denoting classes. A class D may *inherit* from another class C , which means that D has all the fields of C , but may *override* (redefine) some of them and may also have additional fields. In that case, we call C the *superclass* of D , and D a *subclass* of C . When D overrides a field, the new declaration hides the overridden field for D and its subclasses.

An *object* is an instance of a class which has its own identity and state. An object of the class C contains all fields defining C ; we will also say that it is of type C . Its state is given by the value of its attributes and may be modified by applying on it one of the methods defined for C . It can be created by calling a particular method defined in C , called a *constructor*, and destructed by calling another particular method defined in C , called a *destructor*.

A BOON program is a block containing a list of class definitions and a main command using them. So, a class declaration may be either of the form **class** $C\{ \text{attdecl methdecl} \}$, where “attdecl” and “methdecl” are the lists of attribute and method declarations or, if D inherits from C , it may be of the form **class** $D : C\{ \text{attdecl methdecl} \}$.

The attributes may be of two kinds: the standard ones (whose types are subsets of $\mathbb{V}$) or object ones (whose types are classes defined in the program). The corresponding declarations are of the form “**att** $a : V$ ” for a *standard attribute* of type $V \subseteq \mathbb{V}$ and “**oatt** $a : C$ ” for an *object attribute* of type C . The methods are a kind of procedures declared with the clause “**method** $m(\text{parlist}) \text{ block}$ ”, where “parlist” represents a (possibly empty) list of formal parameters.

The declaration of a subclass D of a class C contains only the declarations of overridden and additional fields. We assume that a class has a unique⁶ default constructor (resp. destructor), which just initialises all attributes (resp. releases) at the creation of the object (resp. at the destruction). Thus, the constructor of a class has as many parameters as the class has attributes.

⁶ In fact, it is technically possible to consider user-defined constructors, but for the sake of simplicity and readability, we will consider here only this default constructor.

A standard variable declaration is as before while an object variable is introduced by the clause “**ovar** $c : C$ ”, where c refers to an object of type C . Like a standard variable, an object variable c may appear in an expression as ‘ c ’, c' or c with analogous meaning. We allow *inclusion polymorphism*, which means that an object variable c declared of type C may also refer (at a stage of the execution) to an object of any subclass of C . An attribute a of an object variable c may appear in an expression as $c.a$. Since the actual type κ of the object assigned to c may be a subclass of C , $c.a$ is bound to the attribute a declared in the closest class to D in its inheritance tree, which is determined dynamically at the execution time. This mechanism is called *dynamic binding* and exists also for method calling.

We consider new operations at the expression level, namely “**new** _{C} ”, corresponding to the creation of an object of a class $C \in \mathbb{C}$, and “**del**” corresponding to an object destruction⁷. Typically, they may be used in atomic commands of the form “{ $c' = \text{new}_C(\text{initlist})$ }”, where “initlist” represents the list of initial values for the attributes of the class C , or “{ **del**(obj) }”, where “obj” is an object variable or attribute. Moreover, we consider a new command “ $c.m(\text{arglist})$ ”, which represents the call of the method m on the object c .

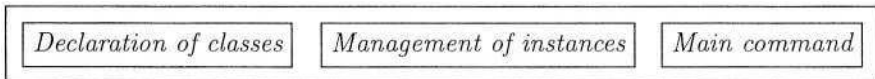
Also, the keywords **super** and **this** can be used in the body of a method of a class C . The keyword **super** is used for referring to the fields of the superclass of C , and may be juxtaposed k times for referring to the fields of the k^{th} parent (assuming that C has at least k parents). For instance **super.super.a** refers to the field a of the superclass of the superclass of C . The keyword **this** is used for referring to the object to which the method is applied. So, in the body of a method m of the class C , an attribute a may appear as **this.a** (and analogously for a method m' of C), which has the same meaning as if **this** was an object variable. The keyword **this** may also be used in an operation at the expression level. For instance, if a method of a class C contains the command

$$\{ d' = \text{new}_D(\text{this}) \},$$

where d is an object variable of type D having an attribute of type C , then if this method is applied to an object c , the execution of the command will initialise the object attribute of d with c .

4 M-net Semantics of Object-Oriented Extensions

Intuitively, the M-net semantics of an object-oriented program involves three parts: the class declarations, class instances management and the main command. All these parts (M-nets) are put in parallel and scoped w.r.t. all communication actions, as sketched below.



⁷ Like in C++, it is allowed but not advisable to call explicitly the destructor.

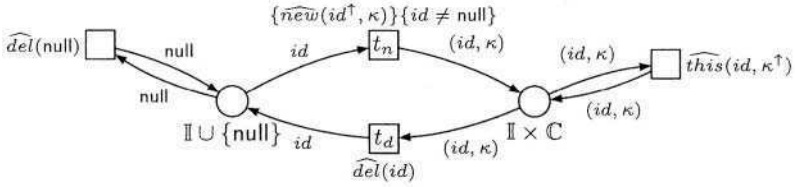


Fig. 2. The internal part of the class instances M-net N_I .

The part for the class declarations is composed of an M-net for each class declaration and one *inheritance directory* M-net. The part devoted to the management of class instances is represented by the *class instances* M-net. As before, the part for the main command is represented by the corresponding M-net.

Except M-nets corresponding to the commands, all M-nets considered from now on are composed of three parts: an entry, an internal and an exit one similarly to the resource M-nets, see Fig. 1. For the sake of readability, we omit the entry and exit parts, which are intended for an adequate initialisation and termination of the internal part. Also, we will omit in the transition labels the empty guards $\{\}$ and the set brackets if the set contains only one element. Moreover, we will signal with an additional arrow “ $\dagger$ ” the action parameters which are intended to be sent (exported), while the remaining parameters are intended to be received, like e.g., in $\widehat{new}(id^\dagger, \kappa)$.

4.1 Management of Instances

Each instance of a class is uniquely identified by an identifier $id \in \mathbb{I}$, which can be considered as a pointer, and has also a *type* κ , which is the class name.

The object identifiers together with their associated types are managed by the class instances M-net N_I , see Fig. 2, which provides a free identifier from the set $\mathbb{I}$ to each new object, keeps its actual type, and gets back each identifier released by a destruction. The size of the set of identifiers $\mathbb{I}$ can be interpreted as the memory size.

We allow the inclusion polymorphism which means that the actual type of an object variable or attribute may be different from its declaration. For instance, if D is a subclass of C , the type of c after execution of the command in the scope “**ovar** $c : C \langle c' = \text{new}_D(\text{initlist}) \rangle$ ” is actually D .

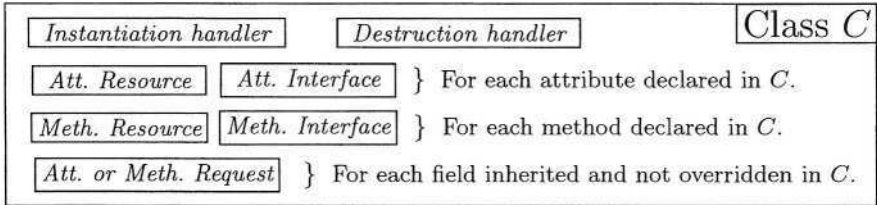
It may also happen that an object variable or attribute is not yet initialised or corresponds to a destructed object. In such a case, this variable or attribute refers to a special (idle) object identified by $\text{null} \notin \mathbb{I}$. The identifier null is never given to an actual object. However, a destruction command applied to a variable or an attribute identified by null is allowed and transparent for the program behaviour.

At the beginning of the execution of the class instances M-net N_I , the place on the left is marked with a token of each value of its type $\mathbb{I} \cup \{\text{null}\}$. The tokens in $\mathbb{I}$ represent the identifiers available for the creations of objects through the action $\widehat{new}(id^\dagger, \kappa)$, while null is used for handling the destruction of idle object variables or attributes through the action $\widehat{del}(\text{null})$. The transition t_n represents

the creation of a new object identified by $id \in \mathbb{I}$ with a given type $\kappa \in \mathbb{C}$. The firing of t_n removes a value corresponding to id from the place on the left and puts in the place on the right the token (id, κ) where κ is the type of the new object identified by id . This place contains at each time all identifiers in use with their associated types; the actual type associated to an identifier id may be checked through the action $\widehat{this}(id, \kappa^\dagger)$. The transition t_d represents the object destruction which releases the identifier id .

4.2 Declarations of Classes

The M-net semantics of a class is mainly composed of an M-net for each standard or object attribute and each method. Each of them has again two parts: the *resource* M-net (which is similar to a $B(PN)^2$ resource M-net) and an *interface* M-net (which is used for handling of dynamic bindings). In order to handle properly the initialisation and the releasing of all the attributes of the instances⁸ of a class, the semantics of a class declaration includes also an *instantiation* and a *destruction* M-net. Moreover, if a class D inherits from a class C , then the semantics of D contains an additional part composed of *request* M-nets corresponding to all attributes and methods which are not declared in D but inherited from C or from an upper class. Each request M-net plays the role of a relay allowing to find the right interface and resource M-nets. Each interface M-net is in fact a terminal request M-net directly linked to an attribute or a method resource M-net. See also the schema below.



Attribute M-nets. The resource M-net of an attribute a keeps the current value of a for each instance of the class where a is declared. If a is a standard attribute, this value belongs to a set $V \subseteq \mathbb{V}$ being the type of a , and if a is an object attribute, it is a pair (id, κ) , where id is the identifier of the object of type κ assigned to a . For instance, if a is an object attribute belonging to a class C , then for each object c of type C identified by id_c , the resource M-net of a carries a token $(id_c, (id, \kappa))$, where (id, κ) is the actual value of the object assigned to $c.a$. Fig. 3 gives two examples of attribute resource M-nets: $N_{S(sa, V, C)}$ for a standard attribute sa of type V in the class C and $N_{O(oa, D, C)}$ for an object attribute oa of (declared) type D in the class C . Initially, in both cases, the places on the left of the attribute resource net contains as many black tokens as there are existing object identifiers, that is $|\mathbb{I}|$, allowing to store a different value of the attribute for each instance of a class.

⁸ Note that “all the attributes” of a class also include attributes inherited from its parent class and all upper classes.

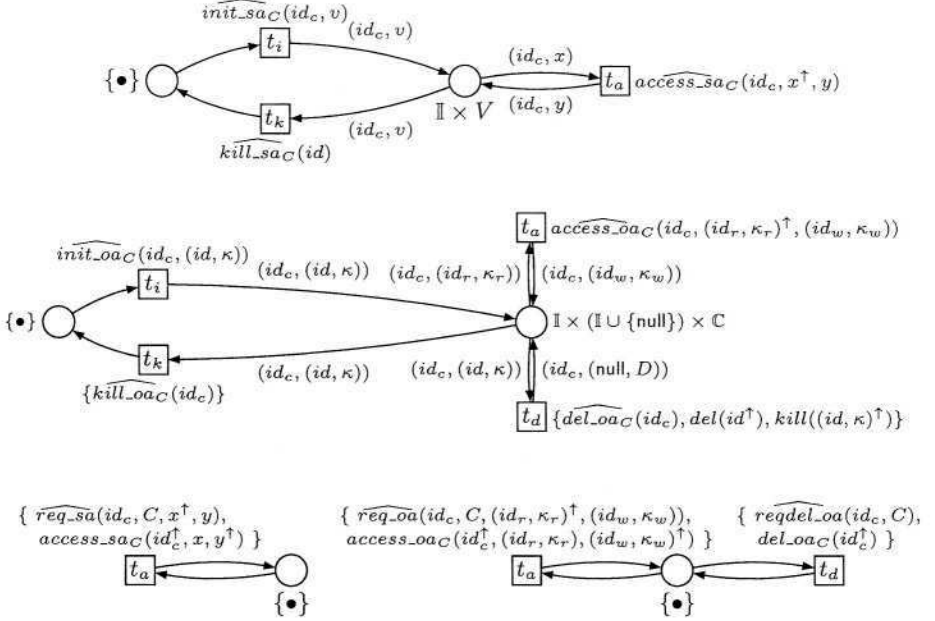


Fig. 3. The resource M-nets $N_{S(sa,V,C)}$ and $N_{O(oa,D,C)}$ and the interface M-nets $N_{SReq(sa,C,C)}$ and $N_{OReq(oa,C,C)}$ of standard and object attributes sa , and oa , resp.

Since an attribute can be redefined in a subclass, we distinguish attributes of different classes but having the same name by quoting this name by the class name; this quoting concerns only actions appearing in resource M-nets and their conjugates, e.g., $\widehat{init\_sa_C}$, $\widehat{access\_sa_C}$ and $\widehat{kill\_sa_C}$ for a standard attribute sa declared in C . The transition t_i is used at the creation of an object of the class C identified by id_c to initialise the attribute sa_C (resp. oa_C). A standard attribute of an object identified by id_c can then be updated through the action $\widehat{access\_sa_C}(id_c, \dots)$ of the transition t_a (similarly to what happens for standard B(PN)² variables). Updating an object attribute concerns its identifier and its type, namely the pair (id_r, κ_r) with r for read values, and (id_w, κ_w) with w for written values. An object attribute may also be explicitly destructed (independently of its owner object) through the transition t_d . Finally, when the owner object is destructed, the associated attributes are also destructed, through the transition t_k . The action $\widehat{kill}(id, \kappa)$ triggers the releasing of all attributes of the object identified by the pair (id, κ) . This releasing is handled by the *destruction* M-net of the class κ , see Fig. 4.

The *interface* M-net of an attribute a is intended to allow subclasses to access the right resource when they did not redefine a . Fig. 3 shows the interface M-net $N_{SReq(sa,C,C)}$ corresponding to a standard attribute sa (first argument) of a class C (second argument) declared in C (third argument) and $N_{OReq(oa,C,C)}$ corresponding to an object attribute oa of a class C declared in C . The action $\widehat{req\_sa}(id_c, C, \dots)$ in $N_{SReq(sa,C,C)}$ (resp. $\widehat{req\_oa}(id_c, C, \dots)$ in $N_{OReq(oa,C,C)}$) is

used for taking into account any access request to the attribute *sa* (resp. *oa*) of the object identified by id_c .

Method M-nets. The resource M-net of a method *m* declared in the class *C* is like a procedure M-net but it takes into account the identifier id_c of the object to which the method is applied. The standard parameters and local variables are handled as for procedures and object ones also in a very similar way, through the action $\widehat{call\_m_C}(id_c, pid, f_1, \dots, f_n)$, where *pid* and $f_1, \dots, f_n$ are as for $B(PN)^2$ procedures, but they may also be object parameters and thus of the form (id, κ) ; *m* is quoted here by the class where it is declared, similarly as for attributes. The corresponding interface M-net is then $N_{MReq(m, C, C)}$, which is like an interface M-net of a standard attribute, transition t_a having the label of the form

$$\{\widehat{req\_m}(id_c, C, f_1, \dots, f_n), call\_m_C(id_c^\uparrow, pid, f_1, \dots, f_n)\}.$$

Instantiation M-net. The creation of a new instance of a class *C* is the result of the execution of an atomic command comprising the operation **new_C**(initlist), see also section 4.3. It involves, in particular, the reservation of a fresh identifier *id* in the instances handler M-net N_I and the initialisation of all the attributes of *C* for the new instance. This is realised by the *instantiation* M-net $N_{Inst(C, \{S(sa_1), \dots, S(sa_i), O(oa_1), \dots, O(oa_j)\})}$ of the class *C*, see Fig. 4, where $sa_1, \dots, sa_i$ and $oa_1, \dots, oa_j$ are the standard and object attributes defined in *C*. The action $new(id, C^\uparrow)$ serves to get a fresh identifier *id* from N_I , the action $\widehat{new_C}(id^\uparrow, v_1, \dots, v_i, (id_1, \kappa_1), \dots, (id_j, \kappa_j))$ allows to get initial values for all the attributes of *C* and to send *id* back to the expression, and all the actions of the form $init\_sa_C(\dots)$ and $init\_oa_C(\dots)$ are used to initialise the attributes of *C*. Initially, the place of the instantiation M-net is marked with $|I|$ black tokens.

If a class *C* inherits from a set of classes $C_1, \dots, C_k$ (*C* inherits from C_1 , which inherits from C_2 , and so on), then its instantiation M-net is denoted:

$$N_{Inst(C, \{S(a_1), \dots, O(a_j)\}, C_1\{S(a_{1-1}), \dots, O(oa_{1-j_1})\}, \dots, C_k\{S(a_{1-k}), \dots, O(oa_{k-j_k})\})},$$

where $a_1, \dots, a_k$ are the attributes defined in *D* and $a_{i-1}, \dots, a_{i-j_i}$ are the attributes defined C_i , $i = 1..k$. This notation is intended for knowing through the name of the net every quote to give to each action initialising the attributes. Note that each of these a_i 's corresponds to a distinct resource M-net.

Destruction M-net. Each time an instance of a class *D* is destructed, the associated attributes are released. The destruction M-net of *D* is intended to release all the attributes bound to the destructed instance of *D*. If *D* has no superclass, its destruction M-net is $N_{Kill(D, \perp, a_1, \dots, a_k)}$, see Fig. 4, where $a_1, \dots, a_k$ are the attributes declared in *D*. If *C* is the superclass of *D*, then the destruction M-net of *D* is $N_{Kill(D, C, a_1, \dots, a_k)}$, which has an additional action $\widehat{kill}(id, C)$ in the label of t_2 , which triggers the releasing of all the attributes inherited from *C* by the object identified by (id, D) . Initially, the place is marked with $|I|$ black tokens.

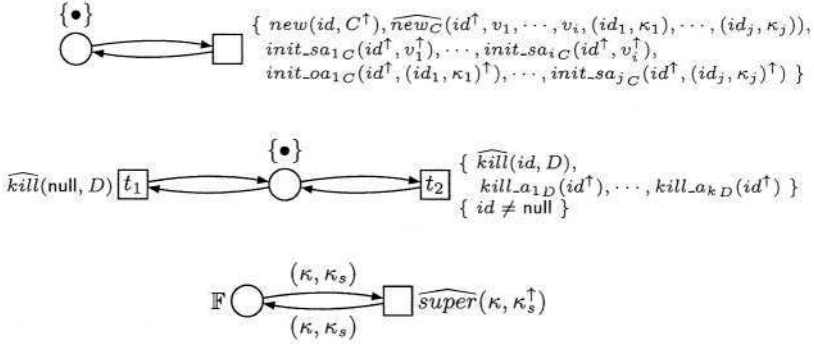


Fig. 4. The M-nets $N_{Inst(C, \{sa_1, \dots, sa_i, oa_1, \dots, oa_j\})}$, $N_{Kill(D, \perp, a_1, \dots, a_k)}$ and $N_{H(F)}$.

Request M-nets. *Request* M-nets are used in a class D in order to handle correctly all the attributes and methods of D inherited from a superclass⁹ C and not overridden in D . They play the role of relays between an access request (to an attribute or a method) and the corresponding interface and resource M-nets, which are actually in the M-nets of C . For instance, if oa is an object attribute declared in C and inherited in D , then the request M-net for oa is $N_{OReq(oa, D, C)}$, which is like an object interface M-net, with the transition t_a having a label of the form

$$\{ \widehat{\text{req\_oa}}(id_c, D, (id_r, \kappa_r)^\dagger, (id_w, \kappa_w)), \text{req\_oa}(id_c^\dagger, C^\dagger, (id_r, \kappa_r), (id_w, \kappa_w)^\dagger) \}.$$

The action $\text{req\_oa}(\dots)$ will synchronise with its conjugate in the corresponding interface M-net, while $\widehat{\text{req\_oa}}(\dots)$ will synchronise with its conjugate in the semantics of a command, see also section 4.3. The request M-nets $N_{SReq(sa, D, C)}$ and $N_{MReq(m, D, C)}$ for a standard attribute sa and for a method m are defined analogously.

Inheritance directory. As explained before, the keyword **super** can be used in order to enforce the access to an attribute or a method of the superclass. Thus, the class inheritance tree must be known all along the execution. It is ensured by the *inheritance directory* M-net $N_{H(F)}$, see Fig. 4. The inheritance tree is stored during the execution of the program in the place of $N_{H(F)}$ as the set of tokens $\mathbb{F} = \{(D, C) \mid C \text{ is the superclass of } D\}$. Notice that for allowing concurrent calls to “**super**” from objects of the same type D , the place must contain as many occurrences of token (D, C) as allowed concurrent calls.

4.3 Commands

The main command is composed as in the original B(PN)², but allows also object variables and some new expressions at the atomic command level. The semantics

⁹ Note that C is not necessarily the immediate superclass of D but is related to it in the inheritance tree.

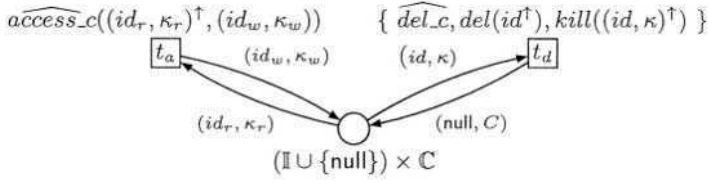


Fig. 5. The object variable resource M-net $N_{(c:C)}$.

of original $B(PN)^2$ commands being unchanged, we address in this section only the semantics of object variables and new kinds of expressions.

Object variables. Like a standard variable, an object variable c of class C is represented by an object variable resource M-net $N_{(c:C)}$ which stores at each time of the execution the pair (id, κ) which identifies the object assigned to c . At the beginning, this value is initialised to $(null, C)$ and may be updated through the action $\widehat{access\_c}((id_r, \kappa_r)^\uparrow, (id_w, \kappa_w))$, which allows to change the current value (id_r, κ_r) into (id_w, κ_w) . The main difference between an object variable and a standard one is that it may explicitly be destructed, releasing the identifier id used for it. At the resource level, it corresponds to the transition t_d , which triggers the releasing of attributes associated to id , releases id and sets the value of the variable to $(null, C)$. An example of an object variable resource M-net is given in Fig. 5. Initially, the place contains the token $(null, C)$.

Method calls and attribute accesses. Method calls and attribute accesses for an object c are modelled in almost the same way, since the corresponding interface M-nets are very similar. In both cases, we need to know the value (id_c, κ_c) of c and use it in a request for the corresponding method or attribute. In the case of a standard attribute sa of c , the access to sa is modelled by the action allowing to read the value of c and the request action $req\_sa(id_c^\uparrow, \kappa_c^\uparrow, x, y^\uparrow)$. The access to an object attribute oa of c is modelled analogously, the request action being $req\_oa(id_c^\uparrow, \kappa_c^\uparrow, (id_r, \kappa_r), (id_w, \kappa_w)^\uparrow)$. For instance, the transition label (lab) of $N_{(lab)}$ being the translation of the atomic BOON command $\langle 'c.sa > 5 \rangle$, which tests whether the attribute sa of c is greater than 5, becomes

$$\{ \widehat{access\_c}((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), \widehat{req\_sa}(id_c^\uparrow, \kappa_c^\uparrow, x, x^\uparrow) \} \{ x > 5 \}.$$

The call of method m is translated to a *call* M-net $N_{Call(lab)}$, where (lab) is the label of the transition devoted to handle m . (lab) contains the action $req\_m(id_c^\uparrow, \kappa_c^\uparrow, e_1^\uparrow, \dots, e_n^\uparrow)$ and all the needed reading actions concerning the effective parameters of m . For instance, the label of the transition handling $c.m(v, d)$, where v is a standard variable and d an object one, becomes:

$$\{ \widehat{access\_c}((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), \widehat{access\_v}(x, x^\uparrow), \\ \widehat{access\_d}((id_d, \kappa_d), (id_d, \kappa_d)^\uparrow), req\_m(id_c^\uparrow, \kappa_c^\uparrow, x^\uparrow, (id_d, \kappa_d)^\uparrow) \},$$

while that of the method call $c.a.m(5)$, where a is an object attribute of c , and m is a method of a is:

$$\{ \text{access\_c}((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), \text{req\_m}(id_a^\uparrow, \kappa_a^\uparrow, 5^\uparrow), \\ \text{req\_a}(id_c^\uparrow, \kappa_c^\uparrow, (id_a, \kappa_a), (id_a, \kappa_a)^\uparrow) \}.$$

Assignments. Assignments to object or standard variables or attributes are modelled in a similar way as for $B(PN)^2$ variables. For an attribute a , it is realised through the action req_a , while for a variable c , through the action access_c . The assignment of an object variable (resp. an object attribute) consists in updating the identifier of the variable (resp. of the object attribute) and its type. For instance, the label of the one-transition M-net corresponding to the command $\langle c.a' = 5 \rangle$ is

$$\{ \text{access\_c}((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), \text{req\_a}(id_c^\uparrow, \kappa_c^\uparrow, x, 5^\uparrow) \}.$$

Object creation and destruction. An object is usually assigned to an object variable or attribute, otherwise it is not reachable. Thus, the class instantiation and object destruction are associated to a variable or attribute access.

Class instantiation. The semantics of a class instantiation $\text{new}_C(\text{initlist})$ consists in creating a new object of the class C and in initialising all the attributes with the corresponding values, through the action

$$\text{new}_C(id, v_1^\uparrow, \dots, v_j^\uparrow, (id_1, \kappa_1)^\uparrow, \dots, (id_j, \kappa_j)^\uparrow),$$

in order to import a fresh object identifier id and to export the initial values of the attributes of the new object. For instance, the label of the one-transition M-net corresponding to the command $\langle c' = \text{new}_C(5, d) \rangle$ where c and d are object variables is

$$\{ \text{access\_c}((id_r, \kappa_r), (id, C)^\uparrow), \text{access\_d}((id_d, \kappa_d), (id_d, \kappa_d)^\uparrow), \\ \text{new}_C(id, 5^\uparrow, (id_d, \kappa_d)^\uparrow), \}.$$

Object destruction. The destruction of an object $\text{del}(c)$ consists in releasing its identifier and the corresponding instance of all its attributes. The releasing of attributes is handled by the destruction M-net of each class. The releasing of the identifier of c is managed by its resource M-net. Thus, for instance, the M-net modelling the atomic BOON command $\langle \text{del}(c) \rangle$ is just a one-transition M-net with the label $\{ \text{del\_c} \}$. For the explicit destruction of an object attribute $c.a$, the label of the corresponding M-net is

$$\{ \text{access\_c}((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), \text{reqdel\_a}(id_c^\uparrow, \kappa_c^\uparrow) \}.$$

The keywords “super” and “this”. In a method body, the keyword “**this**” refers to the object on which the method is applied. Since at its execution, a method carries the identifier of the object on which it is applied, we get the type of this object through the action $this(id, \kappa)$, where κ is to be sent by the class instances M-net N_I . For instance, the transition label corresponding to the command $\langle c.a' = \mathbf{this} \rangle$ in the method applied on an object identified by id is

$$\{this(id^\uparrow, \kappa), access\_c((id_c, \kappa_c), (id_c, \kappa_c)^\uparrow), req\_a(id_c^\uparrow, \kappa_c^\uparrow, (id_a, \kappa_a), (id, \kappa)^\uparrow)\}.$$

The translation of a command involving an attribute or a method of a super-class, represented by the presence of strings **super(.super)**^{*j*}, generates in the labels of corresponding transitions the additional actions

$$super(\kappa_0^\uparrow, \kappa_{s_1}), \dots, super(\kappa_{s_j}^\uparrow, \kappa_{s_{j+1}}),$$

where κ_0 is the class where the command is written. For instance, the transition label corresponding to $\langle \mathbf{super.super.sa} > 5 \rangle$ written in a method of C is

$$\{\widehat{req\_sa}(id_c^\uparrow, \kappa_2^\uparrow, x, x^\uparrow), super(C^\uparrow, \kappa_1), super(\kappa_1^\uparrow, \kappa_2)\}\{x > 5\},$$

where id_c is the identifier of the object to which the method is applied.

5 An Example

This section gives a small example of a BOON program *Prog*, described below, and its translation into M-nets (the destruction is not illustrated here).

```

program
  begin class P {
    att n : {"A", "B", "C"}
    att a : {0..100}
    att w : {1..150}
  }
  class Q : P { att c : {"L", "M", "H"} }
  begin ovar q : Q
     $\langle q' = \mathbf{new}_Q("M", "A", 25, 60) \rangle$ 
    do  $\langle q.w' = 'q.w + 1 \rangle$ ; repeat
       $\square \langle q.w' = 'q.w - 1 \rangle$ ; repeat
      exit
    od
    ;
    do  $\langle 'q.w < 50 \wedge q.c' = "L" \rangle$ ; exit
       $\square \langle 'q.w \geq 50 \wedge 'q.w < 90 \wedge q.c' = "M" \rangle$ ; exit
       $\square \langle 'q.w \geq 90 \wedge q.c' = "H" \rangle$ ; exit
    od
  end
end

```

Two classes are defined: P (Person) and Q (Sportsperson). A Person has a name n , an age a and a weight w , while a Sportsperson is a Person having additionally a weight class c . The translation of the above program $Prog$ is the initially marked M-net:

$$\mathbf{Mnet}(Prog) = (N_I \parallel N_{H(\{Q,P\})} \parallel N_P \parallel N_Q \parallel N_{Com}) \text{ sc } Act ,$$

where N_P , N_Q and N_{Com} are respectively the M-nets of the classe P , the class Q and of the main command, with Act being the set of actions involved in the M-net $\mathbf{Mnet}(Prog)$ comprising new , del , $this$, $super$, new_P , $init\_n_P$, $access\_n_P$, $kill\_n_P$, req_n etc.

The M-net N_P is built as described in the section 4. It is composed of an instantiation and a destruction M-net, a resource M-net for each standard attribute n , a and w , and the interface M-net for each of these resource M-nets:

$$\begin{aligned} N_P = & (N_{Inst}(P, \{S(n), S(a), S(w)\}) \parallel N_{Kill}(P, \perp, S(n), S(a), S(w)) \\ & \parallel N_{S(n, \{^n A^n, ^n B^n, ^n C^n\}, P)} \parallel N_{SReq}(n, P, P) \\ & \parallel N_{S(a, \{0..100\}, P)} \parallel N_{SReq}(a, P, P) \\ & \parallel N_{S(w, \{1..150\}, P)} \parallel N_{SReq}(w, P, P)) \end{aligned}$$

Figure 6 shows the instantiation and destruction M-nets as well as the resource and interface M-nets of the attribute n . The resource and interface M-nets of the remaining attributes are analogous.

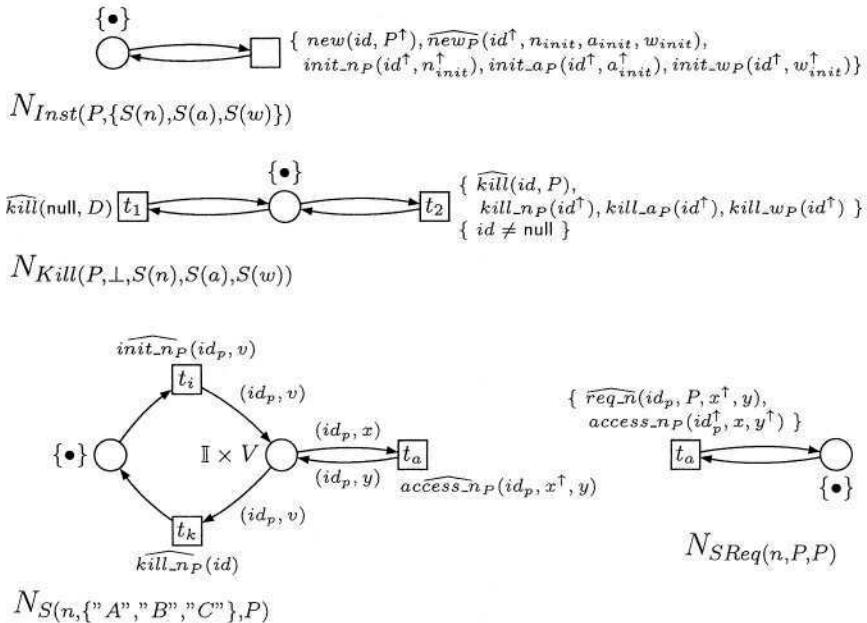


Fig. 6. Some nets composing N_P .

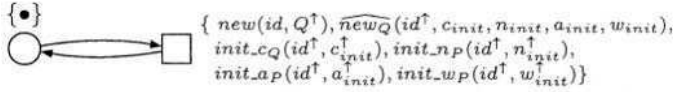


Fig. 7. The Instantiation M-net $N_{Inst}(Q, \{S(c)\}, P\{S(n), S(a), S(w)\})$ of the class Q .

The M-net N_Q of the class Q is built the same way, however, since Q inherits from P , its M-net contains the Resource M-net and the Interface M-net for the new attribute c and the request M-nets for each inherited attributes n, a and w :

$$N_Q = (N_{Inst}(Q, \{S(c)\}, P\{S(n), S(a), S(w)\}) \parallel N_{Kill}(Q, P, c) \\ \parallel N_{S(c), \{ "L", "M", "H" \}, Q} \parallel N_{SReq}(c, Q, Q) \\ \parallel N_{SReq}(n, Q, P) \parallel N_{SReq}(a, Q, P) \parallel N_{SReq}(w, Q, P))$$

The M-net N_{Com} (Fig. 8) of the main command¹⁰ (including the resource M-net $N_{(q:Q)}$ of the object variable q) is composed using binary composition operations of parallel composition ($\parallel$), sequential composition ($;$), choice ($\square$) and iteration ($\otimes$). The first argument of $\otimes$ is placed in a loop and may be iterated zero or more times while its second argument is executed only once and corresponds to an exit statement.

$$N_{Com} = N_{(q:Q)} \parallel \\ (N_{\{access\_q((id_r, \kappa_r), (id, Q)^\dagger), new_Q(id, "M", "A", 25, 60)\}} \{ \} ; \\ (N_{\{access\_q((id, \kappa), (id, \kappa)^\dagger), req\_w(id, \kappa, x, y)\}} \{ y = x + 1 \} \\ \square N_{\{access\_q((id, \kappa), (id, \kappa)^\dagger), req\_w(id, \kappa, x, y)\}} \{ y = x - 1 \}) \otimes N_{\{ \} \{ \} ; \\ N_{\{ \} \{ \} } \otimes \\ (N_{\{access\_q((id, \kappa), (id, \kappa)^\dagger), req\_w(id, \kappa, x, x), req\_c(id, \kappa, y, "L")\}} \{ x < 50 \} \\ \square N_{\{access\_q((id, \kappa), (id, \kappa)^\dagger), req\_w(id, \kappa, x, x), req\_c(id, \kappa, y, "M")\}} \{ 50 \leq x \wedge x < 90 \} \\ \square N_{\{access\_q((id, \kappa), (id, \kappa)^\dagger), req\_w(id, \kappa, x, x), req\_c(id, \kappa, y, "H")\}} \{ x \geq 90 \}))$$

Finally, all these parts (N_I , $N_H(\{Q, P\})$, N_P , N_Q and N_{Com}) are combined (using parallel composition and scoping) producing the semantics of *Prog*.

A new object q of the class Q is created using the action new_Q which is synchronised with $\widehat{new_Q}$ in the instantiation net of Q (Fig. 7). The execution of the corresponding transition allows to get a fresh identifier from the class instance M-net N_I (Fig. 2). This identifier is stored as the value of q in the resource net $N_{(q:Q)}$ and may be updated through the synchronisation on $access\_q$. The other parameters of $\widehat{new_Q}$ are the initial values of the attributes of the class Q which are communicated to their resource M-nets through the synchronisation on the corresponding $init\_...$ actions. In this way, a new instance of each attribute is bound to the newly created object, whose “reference” is kept in the resource M-net of the object variable q .

A statement consisting in reading or updating the attribute $q.w$, e.g. $\langle q.w' = 'q.w + 1 \rangle$, involves first a synchronisation on $access\_q$ which allows to get the

¹⁰ The initialising and termination part are omitted here.

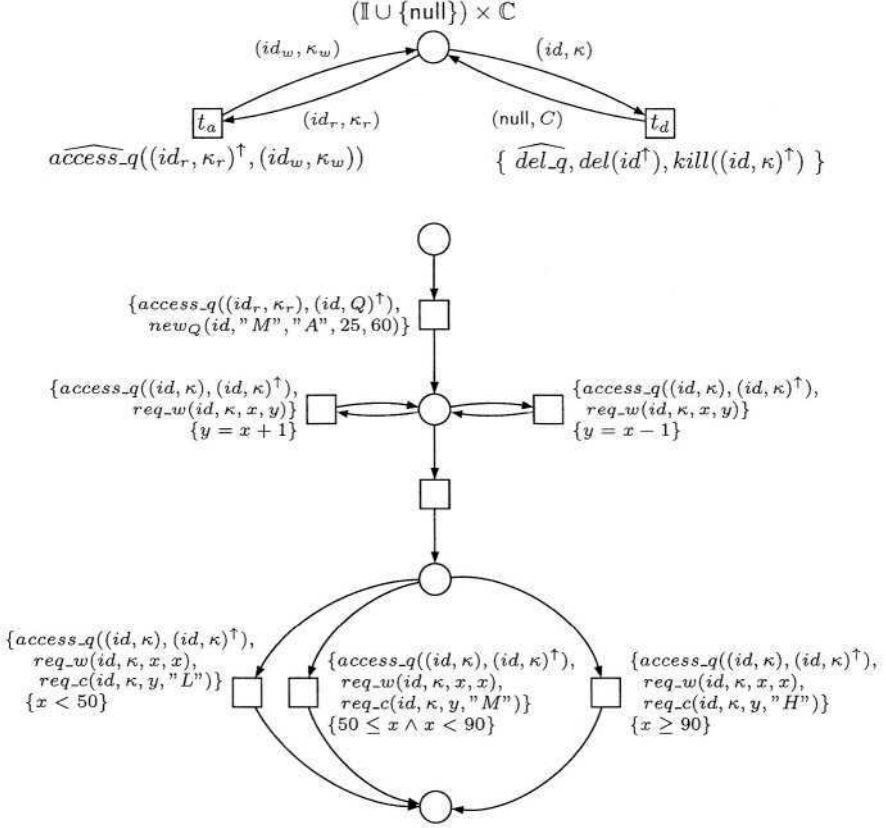


Fig. 8. The components of the M-net of the main command N_{Com} .

identifier and type of the object q . Since the type of q is Q , the reading or updating of w in its resource M-net (in N_P) is obtained by using the request M-net of w in N_Q (through the synchronisation on req_w) and the interface M-net of w in N_P (through the synchronisation on $access\_w_P$). This illustrates the way an inherited attribute can be updated or read.

If w was overridden in the class Q , a new resource M-net w would be in N_Q and the request M-net $N_{SReq(w,Q,P)}$ of w in N_Q would be replaced by an interface M-net $N_{SReq(w,Q,Q)}$. In this case, updating or reading of $q.w$ would be obtained through the synchronisation on req_w and $access\_w_Q$. Also, the attribute w defined in the class P would be reachable from a method of Q by **super**. w . This would involve a synchronisation on *super* with the inheritance directory M-net $N_{H(\{Q,P\})}$.

6 Soundness

In this section we bring together some arguments allowing to support the soundness of our semantics. This concerns mainly the class instantiation, object initialisation, inclusion polymorphism and dynamic bindings.

Actually, as explained in section 4.3, the operation $\mathbf{new}_C(\text{initlist})$ allows to create a new instance of the class C and to initialise all the attributes according to “initlist”. The new instance is uniquely identified by id coming from the class instances M-net N_I . More precisely, getting id is relayed, using the action $\mathbf{new}_C(id, v_{a_1}^\uparrow, \dots)$, by the instantiation M-net $N_{Inst(C, \{S(a_1), \dots, O(a_k)\})}$. It gets id from the M-net N_I using the action $\mathbf{new}(id, C^\uparrow)$ and initialises all the attributes of the new object with corresponding initial values through actions $\mathbf{init}_{a_i}(id^\uparrow, v_{a_i}^\uparrow)$ if a_i is a standard attribute and v_{a_i} is its initial value (resp. $\mathbf{init}_{a_i}(id^\uparrow, (id_{a_i}, \kappa_{a_i})^\uparrow)$ if a_i is an object attribute). The object creation is sound because N_I does not allow to create two distinct objects with the same identifier and $N_{Inst(C, S(a_1), \dots, O(a_k))}$ correctly initialises all attribute resource M-nets. During the execution, the attributes and the methods are always supplied with the identifier of the object they correspond to. Each attribute access and each method call takes the object identifier into account that guaranties a correct handling of them.

The proposed extension supports the inclusion polymorphism, it means that it is possible to assign an object of a subtype of the declared type to an object variable or an object attribute. In such a case, the right attributes and methods must be found at the execution time (dynamic binding), as described in section 4. This is obtained using “relay” M-nets (namely, the request and interface M-nets) between the attribute or method invocation and the corresponding resource M-nets. The inheritance tree is known statically which allows to build correctly the request M-nets w.r.t the corresponding interface M-nets, which is crucial and allows to handle the right attribute or method. An analogous mechanism, but working in the opposite direction, is used in order to access the attributes or methods of the upper classes when the keyword **super** is used.

7 Conclusion

We introduced a new formalism, called BOON (Basic Object-Oriented Notation), devoted to specify concurrent systems using object-oriented concepts. We defined for it a fully compositional Petri net semantics in terms of M-nets (a class of coloured Petri nets provided with a process algebra-like operations). This led to the development of a representation for classes, objects, attributes and methods supporting single inheritance and dynamic bindings.

This approach may easily be extended in order to allow for more polymorphism, for instance, by accepting several user-defined constructors or different methods having the same name. The former can be modelled in our approach using customised instantiation M-nets, while the latter can be obtained by simply preprocessing the homonymous methods in order to give them distinct names. Also, in a related paper, we enriched BOON with encapsulation features allowing to give various access restriction levels to class fields (private, protected, public).

As future works, the designing of a garbage collector would be of interest, since the present version forces the designer to release “manually” every object he instantiated. Also, since our model is dedicated for automatic verification,

the development of a tool supporting BOON is planned. Our interest will also focus on the application of M-nets to the translation of UML [4] diagrams.

Acknowledgements We would like to thank the referees for helpful comments. This research was partly supported by the PROCOPE COMETE project.

References

1. E. Best and R. P. Hopkins. *B(PN)² – a Basic Petri Net Programming Notation*. PARLE'93, LNCS 694, 1993.
2. E. Best, W. Fraczak, R. P. Hopkins, H. Klaudel and E. Pelz. *M-nets²: an Algebra of High-Level Petri Nets, with an application to the semantics of concurrent programming languages*. Acta Informatica, 35. Springer, 1998.
3. O. Biberstein, D. Buchs and N. Guelfi. *Object-Oriented Nets with Algebraic Specifications: The CO-OPN/2 formalism*. Advances in Petri Nets on Object-Orientation, LNCS, Springer, 2000.
4. G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1998.
5. C. Bui Thanh and H. Klaudel. *Encapsulation in an Object-Oriented Notation Based on Modular Petri Nets*. Simulation with Petri nets, ESMc'03, Eurosis, 2003.
6. A. Chizzoni. *CLOWN: CClass Orientation With Nets*. Master degree thesis, Univ. of Milan, 1996.
7. H. Fleischhack and B. Grahmann. *A Petri Net Semantics for B(PN)² with Procedures*. PDSE 1997, IEEE Computer Society, Boston, Ma., 1997.
8. B. Grahmann, and E. Best. *PEP – More than a Petri Net Tool*. TACAS'96, LNCS 1055, Springer, 1996.
9. S. Horstmann and G. Cornell. *Core Java 2*, vol. 1 & 2, Prentice Hall, 1999.
10. H. Klaudel. *Compositional High-Level Petri Net Semantics of a Parallel Programming Language with Procedures*. SCP 41 (2001), Elsevier.
11. C. Lakos. *Object Oriented Modelling with Object Petri Nets*. Advances in Petri Nets, LNCS, Springer, 1997.
12. J. Lilius. *OB(PN)²: An Object Based Petri Net Programming Notation*. Euro-Par '96, LNCS 1123, Springer, 1996.
13. J. Lilius. *OOB(PN)²: An Object-Oriented Petri Net Programming Notation*. The 2nd Workshop on Object-Oriented Programming and Models of Concurrency, 1996.
14. J. Lilius. *OB(PN)²: An Object Based Petri Net Programming Notation*. Concurrent Object-Oriented Programming and Petri Nets, special volume in the Advances in Petri Nets series, LNCS 2001, Springer, 2001.
15. J. Lilius and E. Pelz. *An M-net Semantics for B(PN)² with Procedures*. Proc. of ISCIS'96, Volume I, Middle East Technical University, 1996.
16. M. Mäkelä. *MARIA: modular reachability analyser for algebraic system nets*. On-line manual, <http://www.tcs.hut.fi/maria>, 1999.
17. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
18. B. Stroustrup. *The C++ Programming Language*. Addison Wesley, 1986.
19. *Design/CPN Reference Manual for X-Windows*. Online manual, <http://www.daimi.aau.dk/designCPN/>, 1993.

Specification and Verification of Synchronizing Concurrent Objects

Gabriel Ciobanu and Dorel Lucanu*

“A.I.Cuza” University of Iași, Faculty of Computer Science
{gabriel, dlucanu}@info.uaic.ro

Abstract. We introduce a new specification formalism which we call *hiddenCCS*; hidden algebra is used to specify local goals as objects, and CCS is used to describe global goal of the synchronizing concurrent objects. We extend the object specification with synchronization elements associated with methods of different objects, and we use a CCS coordinating module to describe the interaction patterns of methods invocations. Some results refer to strong bisimulation over the *hiddenCCS* configurations. We investigate how the existing tools BOBJ, CWB, and Maude can be integrated to describe and verify useful properties of the synchronizing concurrent objects. The *hiddenCCS* specifications can be described in the rewriting logic using Maude. Finally we present the first steps towards temporal specifications and verification for *hiddenCCS*.

Keywords: Algebraic specification, integration, concurrent systems, object-oriented specification, hidden algebra, CCS, temporal logics, model checking.

1 Introduction

The complexity and dynamic interaction of software components provide challenging research issues in large system design and verification. Although some specification techniques can support concurrent interaction, they generally cannot scale up for modeling the data and state of complex distributed systems. On the other hand, state-based formalisms such as hidden algebra provide specification techniques for capturing complex data and states; however, they are weak for capturing the interaction aspects of concurrent systems. A possible approach is to integrate two specification techniques. In this paper we investigate the integration between hidden algebra and CCS, introducing a powerful specification technique which we call *hiddenCCS*. We use this name to express that we deal with objects specified in *hidden algebra*, and their interactions are described by *CCS* processes.

The new formal model for concurrent objects is able to capture the relevant dynamics (operational semantics) of the whole system, allowing derivation of

* This paper was partially written in June 2003 while the second author has visited National University of Singapore within the EERSS Programme.

the system properties from those of its objects and model checking tools. The symbiosis of object-oriented algebraic specification and interaction process algebra is given by a simple formal glue provided by some synchronization elements added to hidden algebra and appropriate semantic rules. HiddenCCS extends object-oriented hidden algebra with a CCS coordinating module able to describe the interaction patterns of method invocations. From an object-oriented point of view, we preserve the properties and the expressive power of hidden algebra specification and its hidden logic. From a process algebra point of view, we describe the possible patterns of interaction between objects and preserve the expressive power of CCS and its Hennessy-Milner logic.

Hidden algebra takes as basic the notion of equational behavioral satisfaction: this means that hidden specifications characterize how objects behave in response to a given set of experiments. Hidden algebra is able to handle the most troubling features of large systems, including concurrency, nondeterminism, and local states, as well as the usual features of the object paradigm, including classes, subclasses (inheritance), attributes and methods, in addition to logical variables, abstract data types, and generic modules [7].

CCS is a calculus used to specify how the interactive systems should behave. A CCS process expresses the interaction between subsystems as well as the capability of the system to interact with other systems running concurrently. We may think that a CCS process describes communication scenarios which the designed system should be able to perform. Therefore we use CCS to specify the communication requirements.

We extend the algebraic specification with synchronization elements similar to CCS communication channels; such a channel links the object initiating a method invocation and the corresponding object method. The formal operational semantics of hiddenCCS integrates state transition semantics of hidden algebra and CCS reduction rules by using these synchronization elements. In this way we provide a foundation for complex derivations, and a reasoning system by combining hidden logic and Hennessy-Milner logic. This new specification technique is very flexible, allowing the reuse of both object specifications and coordinating CCS module.

The structure of the paper is as follows. Section 2 presents hidden algebra and BOBJ. Section 3 recalls the main definitions of CCS. In Section 4 we introduce the new hiddenCCS specifications, and present some theoretical results. Section 5 presents the existing software tools for CCS and hidden algebra, presenting Maude as a software framework able to handle hiddenCCS specifications. The first steps towards temporal specifications for hiddenCCS are presented in Section 6. Conclusion and references end the paper.

2 Specification of Objects in Hidden Algebra

We assume that the reader is familiar with algebraic specification. A detailed presentation of hidden algebra can be found in [7,13]. Here we briefly reiterate the main concepts and notations.

A (fixed data) *hidden signature* consists of two disjoint sets: V of *visible* sorts, and H of *hidden* sorts; a many sorted $(V \cup H)$ -signature Σ ; and an $\Sigma|_V$ -algebra D called *data algebra*. Such a hidden signature is denoted by Σ , and its constituents are denoted by $H(\Sigma)$, $V(\Sigma)$, and $D(\Sigma)$ respectively. Given a hidden signature (V, H, Σ, D) , a *hidden Σ -model* is a Σ -algebra M such that $M|_{\Sigma|_V} = D$. This means that the interpretation of each sort $s \in V \cup H$ is a distinct set M_s , and the interpretation of a symbol $f \in \Sigma_{s_1 \dots s_n, s}$ is a function $\llbracket f \rrbracket_M : M_{s_1} \times \dots \times M_{s_n} \rightarrow M_s$. By $M|_{\Sigma|_V}$ we denote the algebra M restricted only to the visible sorts and visible operations. A *hidden Σ -homomorphism* $h : M \rightarrow M'$ is a Σ -homomorphism such that $h|_{\Sigma|_V} = \text{id}_D$.

Given a hidden signature (V, H, Σ, D) and a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma|_V = \Sigma|_V$, a Γ -context for sort s is a term in $\mathcal{T}_\Gamma(\{- : s\} \uplus Z)$ having exactly one occurrence of a special variable $\_$ of sort s . Z is an infinite set of distinct variables. $\mathcal{C}_\Gamma[- : s]$ denotes the set of all Γ -contexts for a sort s . If $c \in \mathcal{C}_\Gamma[- : s]$, then the sort of c , seen as a term, is called the *result sort* of the context c . A Γ -context with visible result sort is called a Γ -experiment. If $c \in \mathcal{C}_\Gamma[- : s]$ with the result sort s' and $t \in \mathcal{T}_\Sigma(X)_s$, then $c[t]$ denotes the term in $\mathcal{T}_\Sigma(\text{var}(c) \cup X)$ obtained from c by substituting t for $\_$. Furthermore, for each hidden Σ -model M , c defines a map $\llbracket c \rrbracket_M : M_s \rightarrow [M^{\text{var}(c)} \rightarrow M_{s'}]$ given by $\llbracket c \rrbracket_M(a)(\vartheta) = a_\vartheta(c)$, where $a_\vartheta(c)$ is the variable assignment $\{- \mapsto a\} \cup \{z \mapsto \vartheta(z) \mid z \in \text{var}(c)\}$. We call $\llbracket c \rrbracket_M$ the *interpretation* of the context c in M .

Given a hidden signature (V, H, Σ, D) , a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma|_V = \Sigma|_V$ and a hidden Σ -model M , the Γ -behavioral equivalence on M , denoted by $\equiv_\Sigma^\Gamma$, is defined as follows: for any sort $s \in V \cup H$ and any $a, a' \in M_s$,

$$a \equiv_\Sigma^\Gamma a' \text{ iff } \llbracket c \rrbracket_M(a)(\vartheta) = \llbracket c \rrbracket_M(a')(\vartheta)$$

for all Γ -experiments c and all $(V \cup H)$ -sorted maps $\vartheta : \text{var}(c) \rightarrow M$.

Given an equivalence $\sim$ on M , an operation $f \in \Sigma_{s_1 \dots s_n, s}$ is *congruent wrt $\sim$* iff $\llbracket f \rrbracket_M(a_1, \dots, a_n) \sim \llbracket f \rrbracket_M(a'_1, \dots, a'_n)$, whenever $a_i \sim a'_i$ for $i = 1, \dots, n$. An operation $f \in \Sigma$ is *Γ -behaviorally congruent wrt M* iff it is congruent wrt $\equiv_\Sigma^\Gamma$. A *hidden Γ -congruence* on M is a $(V \cup H)$ -equivalence on M which is the identity on visible sorts and each operation in Γ is congruent with respect to it.

Theorem 1. [7,13] *Given a hidden signature (V, H, Σ, D) , a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma|_V = \Sigma|_V$ and a hidden Σ -model M , then, Γ -behavioral equivalence is the largest hidden Γ -congruence on M .*

Given a Σ -equation e of the form $(\forall X)t = t'$ in C , a hidden Σ -model M Γ -behaviorally satisfies e if and only if for all $\vartheta : X \rightarrow M$, $\vartheta(t) \equiv_\Sigma^\Gamma \vartheta(t')$ whenever $\vartheta(u) \equiv_\Sigma^\Gamma \vartheta(v)$ for all $u = v$ in C . We write $M \models_\Sigma^\Gamma e$. If E is a set of Σ -equations, then we write $M \models_\Sigma^\Gamma E$ iff $M \models_\Sigma^\Gamma e$ for all e in E .

A *behavioral specification* is a triplet $\mathcal{B} = (\Sigma, \Gamma, E)$ consisting of a hidden Σ -signature, a subsignature $\Gamma \subseteq \Sigma$ such that $\Gamma|_V = \Sigma|_V$ and a set of Σ -equations. We often denote the constituents of $\mathcal{B}$ by $\Sigma(\mathcal{B})$, $\Gamma(\mathcal{B})$ and respectively $E(\mathcal{B})$. The elements of $\Gamma \setminus (\Sigma|_V)$ are called *behavioral operations*. A hidden Σ -model M *behaviorally satisfies* the specification $\mathcal{B}$ iff M Γ -behaviorally satisfies E , that is $M \models_\Sigma^\Gamma E$. We write $M \models \mathcal{B}$ and we say that M is a $\mathcal{B}$ -model. For any equation

e , we write $\mathcal{B} \models e$ iff $M \models \mathcal{B}$ implies $M \models e$. An operation $f \in \Sigma$ is *behaviorally congruent* wrt $\mathcal{B}$ iff f is Γ -behaviorally congruent wrt each $\mathcal{B}$ -model M .

Behavioral specifications can be used to model concurrent objects. The framework for simple objects is the monadic fixed-data hidden algebra [13]: $\mathcal{B}$ specifies a *simple object* iff

1. $H(\mathcal{B})$ has a unique element h called *state sort*;
2. each operation $f \in \Sigma \setminus \Sigma \upharpoonright_V$ is either:
 - a hidden (generalized) constant modeling an initial state, or
 - a *method* $g : hv_1 \dots v_n \rightarrow h$ with $v_i \in V$ for $i = 1, \dots, n$, or
 - an *attribute* $q : hv_1 \dots v_n \rightarrow v$ with $v \in V$ and $v_i \in V$ for $i = 1, \dots, n$.

A *concurrent connection* $\mathcal{B}_1 \parallel \dots \parallel \mathcal{B}_n$ is defined as in [8] where the composite state sort is implemented as tupling. If h_i is the state sort of $\mathcal{B}_i$, then a composite state is a tuple $\langle st_1, \dots, st_n \rangle : \text{Tuple}$ where the state st_i is of sort h_i . For $i = 1, \dots, n$, the projection operations i^* are defined by projection equations $i^*(\langle st_1, \dots, st_n \rangle) = st_i$, together with “tupling equation” $\langle 1^*st, \dots, n^*st \rangle = st$ for each st of sort Tuple .

We assume that all specifications $\mathcal{B}_1, \dots, \mathcal{B}_n$ share the same data algebra. For each component $\mathcal{B}_i$ and f in $\mathcal{B}_i$, we further consider an operation $f.\mathcal{B}_i$ defined by:

$$\begin{aligned} f.\mathcal{B}_i(\langle st_1, \dots, st_n \rangle, \vec{x}) &= f(st_i, \vec{x}) \text{ if } f \text{ is an attribute, and} \\ f.\mathcal{B}_i(\langle st_1, \dots, st_n \rangle, \vec{x}) &= \langle st_1, \dots, f(st_i, \vec{x}), \dots, st_n \rangle \text{ if } f \text{ is a method,} \end{aligned}$$

where by $\vec{x}$ we denote a sequence of the form $x_1, \dots, x_n$.

Proposition 1. [7] 1. If f is a method of $\mathcal{B}_i$, g a method of $\mathcal{B}_j$, and $i \neq j$, then

$$f.\mathcal{B}_i(g.\mathcal{B}_j(st, \vec{y}), \vec{x}) = g.\mathcal{B}_j(f.\mathcal{B}_i(st, \vec{x}), \vec{y})$$

2. If f is an attribute of $\mathcal{B}_i$, g a method of $\mathcal{B}_j$, and $i \neq j$, then

$$f.\mathcal{B}_i(g.\mathcal{B}_j(st, \vec{y}), \vec{x}) = f.\mathcal{B}_i(st, \vec{x}).$$

The first part of Proposition 1 allows us to consider the method $f.\mathcal{B}_i \parallel g.\mathcal{B}_j$ such that $f.\mathcal{B}_i \parallel g.\mathcal{B}_j(st, \vec{x}, \vec{y})$ means the concurrent execution of $f.\mathcal{B}_i(st, \vec{x})$ and $g.\mathcal{B}_j(st, \vec{y})$. If $f.\mathcal{B}_i$ and $g.\mathcal{B}_j$ are in Γ , then $f.\mathcal{B}_i \parallel g.\mathcal{B}_j$ is in Γ as well. The second part says that the components do not share memory. The shared memory can be modeled as a distinct object and the synchronization is realized by the mechanism described below. Note that the new added functions $f.\mathcal{B}_i$ could be non-behavioral in $\mathcal{B}$ even if f is behavioral in $\mathcal{B}_i$.

By *object specification* we mean either a simple object specification, or a conservative extension of a concurrent connection of object specifications. We extend an object specification by adding *elements of synchronization* given by pairs $(a, \bar{a})$ denoting the *necessity* and *availability* of a shared name a . Let $\text{Synch}(\mathcal{B})$ denote the set of the synchronization elements of $\mathcal{B}$. Both a and $\bar{a}$ can be associated with behavioral methods of different objects; the invocation of these methods expresses the necessity and the availability of a , respectively. Let $\text{synch}(g) = a$ denote the fact that the name a is associated with the method g . An element a is called *closed synchronization* for $\mathcal{B}$ iff both a and $\bar{a}$ are present in $\mathcal{B}$. If only one (either a or $\bar{a}$) is present in $\mathcal{B}$, then it is called an *open synchronization* for $\mathcal{B}$.

The BOBJ system (<http://www.cs.ucsd.edu/groups/tatami/bobj/>) is used for behavioral specification, computation, and verification. BOBJ extends OBJ3, supporting behavioral specification and verification, and in particular, providing circular coinductive rewriting with case analysis for conditional equations over behavioral theories. Like OBJ3, BOBJ supports ordinary rewriting for order sorted equational logic (modulo attributes), as well as first order parameterized programming. BOBJ is written in Java and so it can run on a wide range of platforms. We enrich the BOBJ syntax by adding the capability of declaring the elements of synchronization. We present two examples inspired by [12].

Example 1. Agency

An agent is working on an assembly line. He receives jobs on a conveyor belt represented by the port i , and dispatches them after assembly along another conveyor belt represented by $\bar{o}$. A job can be easy, neutral, or difficult. Here we consider a system consisting of two concurrently working agents. We start by specifying the data algebra:

```
dth DATA-AGENT is
  sorts JobGrade ErrJobGrade .
  subsort JobGrade < ErrJobGrade .
  ops easy neut diff : -> JobGrade .
  op none : -> ErrJobGrade .
end
```

An agent is specified as a simple object:

```
bth AGENT is
  sort Agent .
  pr DATA-AGENT .
  op easyJob : Agent -> Agent [synch: iE] .
  op neutJob : Agent -> Agent [synch: iN] .
  op diffJob : Agent -> Agent [synch: iD] .
  op endJob : Agent -> Agent [synch: ~o] .
  op grade : Agent -> ErrJobGrade .
  var A : Agent .
  eq grade(endJob(A)) = none .      eq grade(easyJob(A)) = easy .
  eq grade(neutJob(A)) = neut .     eq grade(diffJob(A)) = diff .
end
```

The attributes of the form `synch: iE` are new; they do not appear in the standard definition of BOBJ, and they specify the synchronization elements associated with the respective methods. Generally, a method having such an attribute is concerned with the availability or the necessity of some resource. The notation $\bar{o}$ is represented in BOBJ by `~o`. The whole system is given by the concurrent connection of two agents:

```
bth AG1 is pr AGENT * (sort Agent to Agent1) . end
bth AG2 is pr AGENT * (sort Agent to Agent2) . end
bth AGENCY is
  pr (AG1 || AG2) * (sort Tuple to Agency) .
end
```

Example 2. Job Shop.

We refine the agency example by adding a hammer and a mallet to the system.

A difficult job can be done only with the aid of the hammer, and a neutral job can be done only with either the hammer or the mallet. Obviously, the hammer and the mallet are shared by the two agents. We first refine the specification of an agent by adding the ability to get/release the hammer/mallet.

```
bth AGENT is
  inc AGENT .
  op getHammer : Agent -> Agent [synch: ~gh] .
  op putHammer : Agent -> Agent [synch: ~ph] .
  op getMallet : Agent -> Agent [synch: ~gm] .
  op putMallet : Agent -> Agent [synch: ~pm] .
  eq grade(getHammer(A)) = grade(A) .
  eq grade(putHammer(A)) = grade(A) .
  eq grade(getMallet(A)) = grade(A) .
  eq grade(putMallet(A)) = grade(A) .
end
```

The hammer and the mallet are specified as follows:

```
bth HAMMER is
  sort Hammer .
  op alloc : Hammer -> Hammer [synch: gh] .
  op release : Hammer -> Hammer [synch: ph] .
  op isAv : Hammer -> Bool .
  var H : Hammer .
  eq isAv(alloc(H)) = false .
  eq isAv(release(H)) = true .
end
bth MALLET is
  sort Mallet .
  op alloc : Mallet -> Mallet [synch: gm] .
  op release : Mallet -> Mallet [synch: pm] .
  *** the rest is similar to HAMMER
end
```

In this example the synchronization elements are gh, ph, gm and pm. AG1 and AG2 are using the new definition for AGENT. The new system is specified as the concurrent connection of the two agents, together with the hammer and the mallet:

```
bth JOBSHOP is
  pr (AG1 || AG2 || HAMMER || MALLET) * (sort Tuple to Jobshop)
end
```

It is not clear how the agents coordinate their activities for a correct use of the resources. Their interaction is specified later by adding a CCS coordinating module.

Hidden logic [13] is a generic name for various logics strongly related to hidden algebra offering sound rules for behavioral reasoning which can be easily automated. BOBJ supports hidden logic by implementing behavioral rewriting, circular coinductive rewriting, automatic cobasis generation, and concurrent connection. Here is a sample showing how BOBJ supports hidden logic:

```
BOBJ> select AGENT .
BOBJ> show cobasis .
The cobasis for Agent is:
```

```

    op grade : Agent -> ErrJobGrade
BOBJ> op init : -> Agent .
BOBJ> eq grade(init) = none .
BOBJ> cred init == endJob(easyJob(init)) .
=====
c-reduce in AGENT : init == endJob(easyJob(init))
using cobasis for AGENT:
op grade : Agent -> ErrJobGrade
-----
result: true
BOBJ> select AGENT .
=====

```

The cobasis is used for proving behavioral equivalence by coinduction [13]. For our AGENT specification, two states A and A' are behaviorally equivalent iff $grade(A) = grade(B)$. Then we define a new state $init$ with $grade(init) = none$, and we prove that $init$ and $endJob(easyJob(init))$ are behaviorally equivalent.

3 Calculus of Communicating Systems

The Calculus of Communicating Systems (CCS) was originally developed by Milner in 1980, and supports the synchronization of interacting processes [11]. CCS provides a minimal formal framework to describe and study synchronizing concurrent processes and various behavioral equivalences. Interaction among processes is established by a matching between complementary ends a and $\bar{a}$ of an arbitrary synchronization channel a . When there are many pairs which can satisfy the matching condition, only a single pair $(a, \bar{a})$ is selected.

We assume a set A of *names*; the elements of the set $\bar{A} = \{\bar{a} \mid a \in A\}$ are called *co-names*, and the elements of the set $\mathcal{L} = A \cup \bar{A}$ are *labels* naming ordinary actions. The function $a \mapsto \bar{a}$ is extended to $\mathcal{L}$ by defining $\bar{\bar{a}} = a$. The standard definition of CCS includes only one special action called *silent action* and denoted by τ , intended to represent the internal synchronization of a system.

The *processes* are defined over the set A of names by the following syntactical rules [12]:

$$P ::= 0 \mid \alpha.P \mid P + Q \mid P|Q \mid \mathbf{new} L P \mid A\langle a_1, \dots, a_n \rangle$$

where P and Q range over processes, α over actions, a_i over names, and A over the process identifiers.

A structural congruence relation is defined over the set of processes. The relation $\equiv$ over the set of processes is called *structural congruence*, and is defined as the smallest congruence which satisfies:

- $P \equiv Q$ if Q can be obtained from P by α -conversion
- $P + 0 \equiv P$, $P + Q \equiv Q + P$, $(P + Q) + R \equiv P + (Q + R)$,
- $P | 0 \equiv P$, $P | Q \equiv Q | P$, $(P | Q) | R \equiv P | (Q | R)$.

The structural operational semantics is shown in Figure 1, where we have already assumed that the summation and parallel composition are associative and commutative. If $\vec{a} = (a_1, \dots, a_n)$ and $\vec{b} = (b_1, \dots, b_n)$, then $\{\vec{b}/\vec{a}\}P$ denotes the renaming construction $P[b_1/a_1, \dots, b_n/a_n]$. We also assume that every process identifier A has a defining equation of the form $A(\vec{a}) \stackrel{\text{def}}{=} P_A$ where P_A is a summation of processes, and $\vec{a}$ includes all the free names in P_A .

$$\begin{array}{c}
 \frac{}{\alpha.P \xrightarrow{\alpha} P} \qquad \frac{P \xrightarrow{\alpha} P'}{P + Q \xrightarrow{\alpha} P'} \\
 \\
 \frac{P \xrightarrow{\alpha} P'}{P|Q \xrightarrow{\alpha} P'|Q} \qquad \frac{\{\vec{b}/\vec{a}\}P_A \xrightarrow{\alpha} P'}{A(\vec{b}) \xrightarrow{\alpha} P'} \quad A(\vec{a}) \stackrel{\text{def}}{=} P_A \\
 \\
 \frac{P \xrightarrow{\alpha} P'}{\text{new } L \ P \xrightarrow{\alpha} \text{new } L \ P'} \quad \alpha \notin L \cup \bar{L} \qquad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\bar{\alpha}} Q'}{P|Q \xrightarrow{\tau} P'|Q'} \\
 \\
 \frac{P \xrightarrow{\tau} P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \xrightarrow{\tau} Q}{P \xrightarrow{\alpha} Q}
 \end{array}$$

Fig. 1. CCS operational semantics rules

Strong bisimulation, written $\sim$, is defined over the processes as the largest symmetrical relation such that: if $P \sim Q$ and $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \sim Q'$.

Weak bisimulation, written $\approx$, is defined over the processes as the largest symmetrical relation such that: if $P \approx Q$ and $P \xrightarrow{\alpha} P'$, then there exists Q' such that $Q \xRightarrow{\alpha} Q'$ and $P' \approx Q'$.

The following two examples show how it is possible to describe patterns of synchronization by using CCS expressions.

Example 3. Agency (continued).

The hidden specification AGENCY does not include any constraint concerning the order in which the methods are executed. Using the synchronization elements, we use CCS expressions to describe scenarios followed by the two agents:

$$A \stackrel{\text{def}}{=} iE.B + iN.B + iD.B, \quad B \stackrel{\text{def}}{=} o.A$$

$$\text{Agency} \stackrel{\text{def}}{=} A \mid A$$

Example 4. Job Shop (continued).

The hidden specification JOBSHOP say nothing about how the two agents are using the hammer and the mallet. We use CCS expressions to specify that two agents cannot use a tool at the same time:

$$\begin{aligned}
H &\stackrel{\text{def}}{=} \text{gh}.H', \quad H' \stackrel{\text{def}}{=} \text{ph}.H, \quad M \stackrel{\text{def}}{=} \text{gm}.M', \quad M' \stackrel{\text{def}}{=} \text{pm}.M, \\
J &\stackrel{\text{def}}{=} \text{iE}.J_E + \text{iN}.J_N + \text{iD}.J_D, \\
J_E &\stackrel{\text{def}}{=} \bar{o}.J, \quad J_N \stackrel{\text{def}}{=} \overline{\text{gh}}.\overline{\text{ph}}.J_E + \overline{\text{gm}}.\overline{\text{pm}}.J_E, \quad J_D \stackrel{\text{def}}{=} \overline{\text{gh}}.\overline{\text{ph}}.J_E \\
\text{JobShop} &\stackrel{\text{def}}{=} \text{new gh ph gm pm } (J \mid J \mid H \mid M)
\end{aligned}$$

The CCS specification *JobShop* describes a pattern of interaction that cannot be described in hidden algebra. It is possible to have various patterns of interaction between objects. For instance, a neutral job can use only the mallet and we have then: $J_N \stackrel{\text{def}}{=} \overline{\text{gm}}.\overline{\text{pm}}.J_E$. In this way we get a clear concern separation: the local goals are given by the object specifications, and the global goal is given by the CCS coordinating process. In the next section we present the semantic integration of these two modules.

Hennessey-Milner Logic (HML) is a primitive modal logic of actions used for describing local capabilities of CCS processes. HML formulas are as follows:

$$\varphi ::= \text{tt} \mid \text{ff} \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid [a]\varphi \mid \langle a \rangle \varphi \mid \llbracket a \rrbracket \varphi \mid \langle\langle a \rangle\rangle \varphi.$$

If P is a CCS process and φ a HML formula, the satisfaction relation $P \models \varphi$ is inductively defined as:

$$\begin{aligned}
P \models \text{tt}, \quad P \models \varphi_1 \wedge \varphi_2 &\text{ iff } P \models \varphi_1 \text{ and } P \models \varphi_2, \quad P \models \varphi_1 \vee \varphi_2 &\text{ iff } P \models \varphi_1 \text{ or } P \models \varphi_2 \\
P \models [a]\varphi &\text{ iff } (\forall P' \in \{P'' \mid P \xrightarrow{a} P''\}) P' \models \varphi \\
P \models \langle a \rangle \varphi &\text{ iff } (\exists P' \in \{P'' \mid P \xrightarrow{a} P''\}) P' \models \varphi \\
P \models \llbracket a \rrbracket \varphi &\text{ iff } (\forall P' \in \{P'' \mid P \xRightarrow{a} P''\}) P' \models \varphi \\
P \models \langle\langle a \rangle\rangle \varphi &\text{ iff } (\exists P' \in \{P'' \mid P \xRightarrow{a} P''\}) P' \models \varphi
\end{aligned}$$

Example 5. Agency (continued).

Here are two examples of HML formulas satisfied by the agent processes:

$$A \models \langle \text{iE} \rangle \text{tt} \wedge \langle \text{iN} \rangle \text{tt} \wedge \langle \text{iD} \rangle \text{tt}, \quad B \models \langle \bar{o} \rangle \text{tt}$$

4 HiddenCCS Specifications

The integration of CCS and object specification in hidden algebra is given by the elements of synchronization. A CCS process over the elements of synchronization works as a coordinating module that manages the interaction between object components. In our formalism, we provide structure to the CCS actions, and it is enough to consider the pure CCS to model the synchronization between distributed objects. The synchronization elements are provided by pairs $(a, \bar{a})$ with each component associated with a method; in this way we have a method-method interaction. We extend this approach in [1] by adding communication between objects.

A *hiddenCCS specification* is a triple $(\mathcal{B}, \mathcal{P}, \mathcal{IC})$ consisting of an object specification $\mathcal{B}$ given in hidden algebra, a CCS description $\mathcal{P}$ of the coordinating module, and a set $\mathcal{IC}$ of integration consistency requirements. The semantics of hiddenCCS specifications is given by a labeled transition system defined over configurations (hidden state, CCS process) as follows:

1. If $P \xrightarrow{\alpha} P'$, then $(st, P) \xrightarrow{\alpha} (st', P')$ where st' is obtained from st by applying the method designated by α .

2. If $P \xrightarrow{\alpha} P'$, $Q \xrightarrow{\bar{\alpha}} Q'$, and α is closed, then $(st, P \mid Q) \xrightarrow{\tau} (st', P' \mid Q')$ where st' is obtained from st by synchronously applying the methods designated by α and $\bar{\alpha}$ whenever integration consistency requirements are satisfied. For instance, the synchronous application of the methods `getHammer` and `alloc`, corresponding to the synchronization given by `ghand` and `gh`, is possible only if `isAv.HAMMER(st) = isAv(3* st) = true`.

This definition is sound if each (co-)name is uniquely associated to a method. Whenever the same name a is related to more than one method, e.g. to the methods $g_1, \dots, g_n$, then we consider n distinct copies $a_1, \dots, a_n$ of the name a , each of them for the corresponding method. Moreover, we define a relation eq given by $a_i \text{eq} a$ for $i = 1, \dots, n$. The operational semantics of CCS is modified as follows. The “synchronization” rule is replaced with:

$$\frac{P \xrightarrow{\alpha_i} P' \quad Q \xrightarrow{\alpha_j} Q'}{P \mid Q \xrightarrow{\tau(\alpha_i, \alpha_j)} P' \mid Q'} \quad \alpha_i \text{eq} \alpha \text{ and } \alpha_j \text{eq} \bar{\alpha}$$

For the silent action τ we use a more exact notation $\tau(\alpha_i, \alpha_j)$ indicating the names involved in such an internal action. This notation is necessary to integrate CCS semantics with the behavioral semantics of hidden algebra. Since τ is used in the definition of the CCS bisimulation, the following rules restore it from $\tau(\alpha_i, \alpha_j)$, where $\alpha^* = \text{if } (\alpha = \tau(\alpha_i, \alpha_j)) \text{ then } \tau \text{ else } \alpha$:

$$\frac{P \xrightarrow{\tau(\alpha_i, \alpha_j)} Q}{P \xrightarrow{\tau} Q} \quad \frac{P \xrightarrow{\tau} P' \quad P' \xrightarrow{\alpha} Q' \quad Q' \xrightarrow{\tau} Q}{P \xrightarrow{\alpha^*} Q}$$

A $\text{CCS}(\mathcal{B})$ -process is a CCS process built over the set $\text{Synch}(\mathcal{B})$. An *integration consistency requirement* expresses the availability of a synchronization resource and consists of a finite set of equations $q(-, d_1, \dots, d_n) = d$, where q is an attribute in Γ and $d, d_1, \dots, d_n \in D(\mathcal{B})$. Let $ic(\alpha_i, \alpha_j)$ denote the integration consistency requirement corresponding to the synchronization pair (α_i, α_j) , where $\alpha_i \text{eq} \alpha$ and $\alpha_j \text{eq} \bar{\alpha}$. A state st of a model M satisfies $ic(\alpha_i, \alpha_j)$ whenever $\llbracket q \rrbracket_M(st, d_1, \dots, d_n) = d$ for each equation $q(-, d_1, \dots, d_n) = d$ of $ic(\alpha_i, \alpha_j)$; we write $st \models ic(\alpha_i, \alpha_j)$.

Given an object specification $\mathcal{B}$ and a hidden $\mathcal{B}$ -model M , we denote by $LTS_{\Gamma, \text{CCS}}(M)$ the labeled transition system defined by the rules of Figure 2, where P, P' , and Q, Q' are $\text{CCS}(\mathcal{B})$ -processes. If st is a state in M and P is a CCS process, then $LTS_{\Gamma, \text{CCS}}(M, st, P)$ denotes the subsystem induced by the subset of the configurations which are reachable from (st, P) .

In $LTS_{\Gamma, \text{CCS}}(M)$ we have three types of transitions: those corresponding to open synchronizations (labeled by $\alpha = \text{synch}(g)$), those corresponding to closed synchronizations (labeled by τ), and those corresponding to non-synchronizing behavioral methods (labeled by *idle*).

Definition 1. Let Σ be an object signature, Γ a hidden subsignature of Σ , and let M and M' be two Σ -models. The relation $\simeq_{M, M'}^{\Sigma, \Gamma} \subseteq M_h \times M'_h$ is defined by:

$$st \simeq_{M, M'}^{\Sigma, \Gamma} st' \text{ iff } \llbracket q \rrbracket_M(st, d_1, \dots, d_n) = \llbracket q \rrbracket_{M'}(st', d_1, \dots, d_n) \\ \text{for any attribute } q \in \Gamma, d_1, \dots, d_n \in D(\mathcal{B}).$$

$$\begin{array}{c}
\frac{g : h_i \vec{v} \rightarrow h_i, \text{synch}(g) = \alpha, st' = \llbracket g.B_i \rrbracket_M(st, \vec{d}) \quad P \xrightarrow{\alpha} P'}{(st, P) \xrightarrow{\alpha} (st', P')} \\
\\
\frac{g : h_i \vec{v} \rightarrow h_i \in \Gamma \text{ non-synchronizing}, st' = \llbracket g.B_i \rrbracket_M(st, \vec{d})}{(st, P) \xrightarrow{\text{idle}} (st', P)} \\
\\
\frac{g : h_i \vec{v} \rightarrow h_i, \text{synch}(g) = \alpha_i, g' : h_j \vec{v}' \rightarrow h_j, \text{synch}(g') = \alpha_j, \\ st \models \text{ic}(\alpha_i, \alpha_j), st' = \llbracket g.B_i \rrbracket_M \llbracket g'.B_j \rrbracket_M(st, \vec{d}, \vec{d}'), P \xrightarrow{\tau(\alpha_i, \alpha_j)} P'}{(st, P) \xrightarrow{\tau} (st', P')} \\
\\
\frac{(st_1, P) (\xrightarrow{\tau} \cup \xrightarrow{\text{idle}})^* (st'_1, P') \quad (st'_1, P') \xrightarrow{\alpha} (st'_2, Q') \quad (st'_2, Q') (\xrightarrow{\tau} \cup \xrightarrow{\text{idle}})^* (st_2, Q)}{(st_1, P) \xrightarrow{\alpha} (st_2, Q)}
\end{array}$$

Fig. 2. The transition systems associated with a hiddenCCS specification

Definition 2. Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, and two $\mathcal{B}$ -models M and M' , then the behavioral CCS-based strong Γ -bisimulation between M and M' is the largest relation $\sim_{M, M'}$ such that $(st_1, P) \sim_{M, M'} (st'_1, P')$ implies

1. $st_1 \simeq_{M, M'}^{\Sigma, \Gamma} st'_1$,
2. if $(st_1, P) \xrightarrow{\alpha} (st_2, Q)$, then there is (st'_2, Q') such that $(st'_1, P') \xrightarrow{\alpha} (st'_2, Q')$ and $(st_2, Q) \sim_{M, M'} (st'_2, Q')$, and
3. if $(st'_1, P') \xrightarrow{\alpha} (st'_2, Q')$, then there is (st_2, Q) such that $(st_1, P) \xrightarrow{\alpha} (st_2, Q)$ and $(st_2, Q) \sim_{M, M'} (st'_2, Q')$.

Not all the configurations are acceptable for execution. For instance, a configuration $(\text{init}, \text{JobShop})$ is not acceptable if $\text{isAv.HAMMER}(\text{init}) = \text{false}$, because $(\text{init}, \text{JobShop}) \xrightarrow{\text{id}} (st', P')$, st' does not satisfy $\text{ic}(\text{gh}, \overline{\text{gh}})$, and P' requires this integration consistency. We say that a state st is *consistent* with a CCS process P iff for each configuration $(st', P') \in \text{LTS}_{\Gamma, \text{CCS}}(M, st, P)$, st' satisfies the integration consistencies required by P' . A ground term t of state sort is *consistent* with a process P iff $\llbracket t \rrbracket_M$ is consistent with P for each $\mathcal{B}$ -model M .

Proposition 2. Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, two $\text{CCS}(\mathcal{B})$ -processes P and P' , and two $\mathcal{B}$ -models M and M' , then $P \sim P'$ whenever there are st in M and st' in M' such that st is consistent with P , st' is consistent with P' , and $(st, P) \sim_{M, M'} (st', P')$.

Proof. Let us consider $(st, P) \sim_{M, M'} (st', P')$ such that st is consistent with P and st' is consistent with P' . We have to show that $P \sim P'$. Let R be the relation defined by $Q R Q'$ iff there are st_1 and st'_1 such that (st_1, Q) is reachable from (st, P) , (st'_1, Q') is reachable from (st', P') , and $(st_1, Q) \sim_{M, M'} (st'_1, Q')$. We show that if $Q_1 R Q'_1$ and $Q_1 \xrightarrow{\alpha} Q_2$, then there is Q'_2 such that $Q_2 \xrightarrow{\alpha} Q'_2$ and $Q_2 R Q'_2$. It follows that $R \subseteq \sim$.

Proposition 3. *Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ and a $\mathcal{B}$ -model M , then $(st, P) \sim_{M,M} (st', P')$ whenever $st \equiv_{\Sigma}^{\Gamma} st'$ and $P \sim P'$.*

Proof. We define the relation R by $(st, P) R (st', P')$ iff $st \equiv_{\Sigma}^{\Gamma} st'$ and $P \sim P'$. If $(st, P) R (st', P')$ then $st \simeq_{M,M}^{\Sigma, \Gamma} st'$, because $\simeq_{M,M}^{\Sigma, \Gamma}$ is included in the behavioral equivalence. We show that if $(st_1, P_1) R (st'_1, P'_1)$ and $(st_1, P_1) \xrightarrow{\alpha} (st_2, P_2)$, then there are st'_2 and P'_2 such that $(st'_1, P'_1) \xrightarrow{\alpha} (st'_2, P'_2)$ and $(st_2, P_2) R (st'_2, P'_2)$. Therefore $R \subseteq \simeq$.

The strong bisimulation $\sim_{M,M'}$ is able to avoid the experiments that are not meaningful from the interaction viewpoint. Since $\equiv_{\Sigma}^{\Gamma}$ is not designed to do this, the converse of Proposition 3 is not generally true. Here is a counterexample.

Example 6. Buffer.

```

dth DATA is sort Data .   inc NAT .
  subsort Nat < Data .
  op err : -> Data .
end
bth BUFF is sort Buff .   inc DATA .
  op put : Buff Nat -> Buff [synch: in] .
  op del : Buff -> Buff [synch: ~out] .
  op get : Buff -> Data .
  var N : Nat . var B : Buff .
  eq get(del(B)) = err .
  eq get(put(B, N)) = N if get(B) == err .
end

```

Consider $\Gamma = \text{Sig}(\text{BUFF})$ and the CCS processes $B \stackrel{\text{def}}{=} \text{in}.B'$, $B' \stackrel{\text{def}}{=} \overline{\text{out}}.B$. If init is a state with $\text{get}(\text{init}) = \text{err}$ and $st = \text{del}(\text{put}(\text{init}, n))$, then init and st are not behaviorally equivalent, but we have $(\text{init}, B) \sim_{M,M} (st, B)$ and $(\text{init}, B') \sim_{M,M} (st, B')$ in any BUFF-model M . These two states are not behaviorally equivalent because it may exist a model where the result of the experiment $\text{get}(\text{put}(\text{put}(\_, m), n))$ is unpredictable, e.g., $\text{get}(\text{put}(\text{put}(\text{init}, m), n)) = n$ and $\text{get}(\text{put}(\text{put}(st, m), n)) = \text{err}$. This happens because the method $\text{put}()$ is underspecified in the hidden specification BUFF. If we add the CCS coordinating module, then $\text{put}()$ becomes completely specified in the resulting hiddenCCS specification since the CCS module allows only defined experiments [7] and it avoids repeating $\text{put}()$ actions.

Behavioral CCS-based strong Γ -bisimulation can be extended to different specifications related by signature morphisms.

Definition 3. *Given two object specifications $\mathcal{B} = (\Sigma, \Gamma, E)$ and $\mathcal{B}' = (\Sigma', \Gamma', E')$ with $\text{Synch}(\mathcal{B}) = \text{Synch}(\mathcal{B}')$ and $D(\mathcal{B}) = D(\mathcal{B}')$, a signature morphism $\phi : \Sigma \rightarrow \Sigma'$ preserving behavioral operations and synch, a $\mathcal{B}$ -model M and a $\mathcal{B}'$ -model M' , then the behavioral CCS-based weak Γ -bisimulation between M and M' is the largest relation $\approx_{M,M'}$ such that $(st_1, P) \approx_{M,M'} (st'_1, P')$ implies*

1. $st_1 \simeq_{M,M'}^{\Sigma, \Gamma} st'_1$,
2. if $(st_1, P) \xrightarrow{\alpha} (st_2, Q)$, then there is (st'_2, Q') such that $(st'_1, P') \xRightarrow{\alpha} (st'_2, Q')$ and $(st_2, Q) \approx_{M,M'} (st'_2, Q')$, and

3. if $(st'_1, P') \xrightarrow{\alpha} (st'_2, Q')$, then there is (st_2, Q) such that $(st_1, P) \xrightarrow{\alpha} (st_2, Q)$ and $(st_2, Q) \approx_{M, M'} (st'_2, Q')$.

Example 7. Agency and Job Shop.

Let ϕ be the signature morphism $\phi : \text{AGENCY} \hookrightarrow \text{JOB SHOP}$, M an AGENCY-model, and M' a JOB SHOP-model. If st is a state in M such that

$$\llbracket \text{grade.AG1} \rrbracket_M(st) = \llbracket \text{grade.AG2} \rrbracket_M(st) = \text{none},$$

and st' is a state in M' such that

$$\llbracket \text{grade.AG1} \rrbracket_{M'}(st') = \llbracket \text{grade.AG2} \rrbracket_{M'}(st') = \text{none},$$

then:

$$(st, A \mid A) \approx_{M, M'} (st', \text{new } \vec{t}(J \mid J \mid H \mid M))$$

$$(\llbracket \text{diffJob.AG1} \rrbracket_M(st), A \mid B) \approx_{M, M'}$$

$$(\llbracket \text{diffJob.AG1} \rrbracket_{M'}(st'), \text{new } \vec{t}(J \mid J_D \mid H \mid M))$$

$$(\llbracket \text{diffJob.AG1} \rrbracket_M(st), A \mid B) \approx_{M, M'}$$

$$(\llbracket \text{AG1.getHammer} \rrbracket_{M'}(st'_1), \text{new } \vec{t}(J \mid \overline{\text{ph}}.J_E \mid H' \mid M))$$

$$(\llbracket \text{diffJob} \rrbracket_M(st), A \mid B) \approx_{M, M'}$$

$$(\llbracket \text{putHammer} \rrbracket_{M'}(st'_2), \text{new } \vec{t}(J \mid J_E \mid H \mid M))$$

where $st'_1 = \llbracket \text{diffJob} \rrbracket_{M'}(st')$, $st'_2 = \llbracket \text{getHammer.AG1} \rrbracket_{M'}(st'_1)$.

Proposition 4. *Let us consider two object specifications $\mathcal{B}$ and $\mathcal{B}'$ with $\text{Synch}(\mathcal{B}) = \text{Synch}(\mathcal{B}')$ and $D(\mathcal{B}) = D(\mathcal{B}')$, and a signature morphism $\phi : \Sigma(\mathcal{B}) \rightarrow \Sigma(\mathcal{B}')$ preserving behavioral operations and synch. If P is a $\text{CCS}(\mathcal{B})$ -process, P' is a $\text{CCS}(\mathcal{B}')$ -process, M is a $\mathcal{B}$ -model, and M' is a $\mathcal{B}'$ -model, then $P \approx P'$ whenever there are st in M and st' in M' such that st is consistent with P , st' is consistent with P' , and $(st, P) \approx_{M, M'} (st', P')$.*

Proof. Let us consider $(st, P) \approx_{M, M'} (st', P')$ such that st is consistent with P and st' is consistent with P' . We have to show that $P \approx P'$. Let R be the relation defined by $Q R Q'$ iff there are st_1 and st'_1 such that (st_1, Q) is reachable from (st, P) , (st'_1, Q') is reachable from (st', P') , and $(st_1, Q) \approx_{M, M'} (st'_1, Q')$. We show that if $Q_1 R Q'_1$ and $Q_1 \xrightarrow{\alpha} Q_2$, then there is Q'_2 such that $Q_2 \xrightarrow{\alpha} Q'_2$ and $Q_2 R Q'_2$. It follows that $R \subseteq \approx$.

5 Integrating the Existing Software Tools

5.1 Concurrency Workbench

The Concurrency Workbench(CWB) is an automatic verification tool for finite-state systems expressed in process algebra [4]. It is mainly an extensible tool for verifying systems written in the process algebra CCS. It provides different methods, such as equivalence checking, model checking, simulation and abstraction mechanisms for analyzing system behavior. Even though its utility has been

demonstrated with several case studies, the impact on system design practice of such tools has been limited.

CWB supports the computation of numerous different semantic equivalences and preorders. It also includes a flexible model-checking facility for determining when a process satisfies a formula of an expressive temporal logic, namely the propositional μ -calculus. CWB analyzes systems by converting their CCS specifications into finite labeled transition systems (LTS), and then invoking various routines on these LTSs. The CWB uses an “on-the-fly” approach to construct LTSs from specifications, with transitions of components calculated and then combined appropriately into transitions for the composite system.

Suppose we have a CCS description of our Job Shop example that is stored in a file named `jobshop.ccs`. We present a session with the North Carolina CWB (<http://www.cs.sunysb.edu/~cwb/>):

```
cwb-nc> load jobshop.ccs
```

CWB builds an LTS modeling the behavior of each process, and the `size` command displays the number of states and transitions of the LTS.

```
cwb-nc> size Jobshop
States: 35
Transitions: 104
```

We can check whether *Agency* and *Jobshop* are strong bisimilar (using `eq -S bisim`), or weak bisimilar (using simply `eq`). CWB tells that they are weak bisimilar, but they are not strong bisimilar.

```
cwb-nc> eq -S bisim Agency Jobshop
...
States: 40
Transitions: 126
FALSE...
Agency satisfies:
    <iD>[t]ff
Jobshop does not.
cwb-nc> eq Agency Jobshop
States: 40
Transitions: 126
TRUE
```

When we enter the CWB simulator, we have various choices; typing the number of the selected choice, the simulation proceeded from there. This is similar to the `search` command in Maude. The `random` command lets the simulator make a certain number of random choices from those possible at every successive state in the simulation.

```
cwb-nc> sim Jobshop
cwb-nc-sim> 1
(J | JD | H | M) \ {gh,ph,gm,pm}
1: -- t --> (J | 'ph.JE | BH | M) \ {gh,ph,gm,pm}
2: -- iD --> (JD | JD | H | M) \ {gh,ph,gm,pm}
3: -- iN --> (JN | JD | H | M) \ {gh,ph,gm,pm}
4: -- iE --> (JE | JD | H | M) \ {gh,ph,gm,pm}
cwb-nc-sim> 2
```

```
(JD | JD | H | M) \ {gh,ph,gm,pm}
1: -- t --> ('ph.JE | JD | BH | M) \ {gh,ph,gm,pm}
2: -- t --> (JD | 'ph.JE | BH | M) \ {gh,ph,gm,pm}
```

We can invoke the model checker to determine whether or not an agent satisfies a temporal formula. The currently supported temporal logics for the specification of formulas are the μ -calculus (which is the default) and GCTL. An option indicates that model-checking is to be on-the-fly, if possible. Global model-checking (if possible) is the default. However, global model-checking is not possible for GCTL formulas and is only available for alternation-free μ -calculus formulas.

```
cwb-nc> chk -L gctl Jobshop "E F ~{-}"
Model checking completed.
FALSE, the agent does not satisfy the formula.
Execution time (user,system,gc,real):(0.080,0.000,0.000,0.113)
```

```
cwb-nc> fd Jobshop
States explored: 35
No deadlock found.
Execution time (user,system,gc,real):(0.020,0.000,0.000,0.022)
```

The command `fd` looks for a deadlock of the system described by *JobShop*.

5.2 Maude

Maude [3] is a system supporting both rewriting logic specification and computation. Like BOBJ, Maude has been influenced by the OBJ3 language, which can be regarded as an equational logic sublanguage. Maude supports membership equational logic and rewriting logic computation, which are not supported by BOBJ. On the other hand, the current version of Maude does not support the hidden logic used by BOBJ for behavioral specification and verification. It has good properties as a general semantic framework for giving executable semantics to a wide range of languages and models of concurrency. In particular, it supports the representation of CCS and HML [16].

In this section we show that the transition system associated with a hiddenCCS specification can be naturally represented by a Maude rewrite specification. We first modify the Maude module implementing the operational semantics of CCS by replacing the rule for synchronization with the following two rules:

```
cr1 P | Q => {tau(L)}(P' | Q') if P => {L}P' /\ Q => {~ M}Q' /\ L eq M .
cr2 P | Q => {tau(M)}(P' | Q') if P => {L}P' /\ Q => {~ M}Q' /\ L eq M .
```

where $L \text{ eq } M$ represents the implementation of the `_eq_` relation. Then we change the definition of τ , representing it by a function having as argument one of the actions involved in synchronization. This is not a restriction, because $P | Q \xrightarrow{\tau(\alpha_i, \alpha_j)} P' | Q'$ is the same as $P | Q \xrightarrow{\tau(\alpha_i)} P' | Q'$ and $P | Q \xrightarrow{\tau(\alpha_j)} P' | Q'$. Moreover, we have $\tau(\bar{\alpha}) = \tau(\alpha)$.

Here is the Maude description of the CCS expression *JobShop*:

```
mod JOBSHOP-CCSPROC is
  inc CCS .
  ops gh ph gh1 ph1 gh2 ph2 gm pm gm1 pm1 gm2 pm2 : -> Label .
```

```

ops iE1 iN1 iD1 o1 iE2 iN2 iD2 o2 : -> Label .
ops H HH M MM J1 JE1 JN1 JD1 J2 JE2 JN2 JD2 : -> ProcessId .
op JobShop : -> ProcessId .
eq gh eq gh1 = true .      eq gh eq gh2 = true .
eq ph eq ph1 = true .      eq pm eq ph2 = true .
eq gm eq gm1 = true .      eq gm eq gm2 = true .
eq pm eq pm1 = true .      eq pm eq pm2 = true .
eq context = ( H =def gh . HH ) & ( HH =def ph . H ) &
              ( M =def gm . MM ) & ( MM =def pm . M ) &
              ( J1 =def iE1 . JE1 + iN1 . JN1 + iD1 . JD1 ) &
              ...
              ( JD2 =def ~ gh2 . ~ ph2 . JE2 ) &
              ( JobShop =def ( J1 | J2 | H | M )
                \ gh \ gh1 \ gh2 \ ph \ ph1 \ ph2
                \ gm \ gm1 \ gm2 \ pm \ pm1 \ pm2 ) .

endm

```

gh1, gh2, ph1, ph2, gm1, gm2, pm1, and pm2 are copies of gh, ph, gm, and respectively pm corresponding to the two agents. Note that the operator new $L P$ is represented by the restriction $P \setminus L$. We may use the Maude implementation of HML to verify properties of the CCS processes:

```

Maude> red 'JobShop.ProcessId |= < 'iE1.Label > tt .
reduce in MODAL-LOGIC : 'JobShop.ProcessId |= < 'iE1.Label > tt .
result Bool: true
Maude> red 'JobShop.ProcessId |= < 'gh.Label > tt .
reduce in MODAL-LOGIC : 'JobShop.ProcessId |= < 'gh.Label > tt .
result Bool: false

```

The module MODAL-LOGIC describes HML. Since it is written at the metalevel, the CCS terms are quoted.

The current version of Maude does not yet support behavioral membership logic [10]. The price we pay is that we lose support for verification of the behavioral properties. The *synch* attributes and the rules in Figure 2 are used to build the rewrite rules. Here is (a sketch of) the Maude description of the hiddenCCS specification corresponding to the Job Shop example:

```

mod JOBSHOP-CCS is
  pr JOBSHOP-TEST .
  pr MODAL-LOGIC .
  sort CcsJobshop .
  op <_',_> : Jobshop Term -> CcsJobshop .
  op 1*_ : CcsJobshop -> Jobshop .
  op 2*_ : CcsJobshop -> Term .
  ...
  cr1 [iE1] : < A, P > => < < easyJob(1* A), 2* A, 3* A, 4* A >,
                      succ(P, 'iE1.Label) >
                      if P |= < 'iE1.Label > tt .
  ...
  cr1 [tau] : < A, P > => < < getHammer(1* A), 2* A, alloc(3* A), 4* A >,
                      succ(P, 'tau['gh1.Label]) >
                      if P |= < 'tau['gh1.Label] > tt /\ isAv(3* A) .
  ...
endm

```

HML is used in the conditional part of the rewriting rules to select the appropriate action. The function `succ` returns the successor of the process given as the first argument, obtained after applying the transition given by the second argument. Note that the successor is unique for *JobShop* processes. Generally, when there exist more successors, it is necessary to include a rule for each successor.

Once we have a Maude description of the hiddenCCS specification, we can use rewriting logic and its specific Maude commands to check some properties of the specification. For instance, if we consider the following initial state:

```
mod JOBSHOP-TEST is
  pr JOBSHOP .
  op init : -> Jobshop .
  eq grade(1* init) = none .      eq grade(2* init) = none .
  eq isAv(3* init) = true .       eq isAv(4* init) = true .
endm
```

then we can use the command `search` to see the possible transitions from `(init, JobShop)`:

```
Maude> search < init, 'JobShop.ProcessId > =>1 ST:CcsJobshop .
Solution 1 (state 1)
ST:CcsJobshop --> < < easyJob(1* init),2* init,3* init,4* init >,
'[_][_]'H.ProcessId,'M.ProcessId,'J1.ProcessId,'J2.ProcessId],
'gh.Label], 'gh1.Label], 'gh2.Label], 'ph.Label], 'ph1.Label], 'ph2.Label],
'gm.Label], 'gm1.Label], 'gm2.Label], 'pm.Label], 'pm1.Label], 'pm2.Label] >
...
Solution 6 (state 6)
ST:CcsJobshop --> < < 1* init,diffJob(2* init),3* init,4* init >,
...
No more solutions.
```

Note the prefixed notation used for processes. For instance, the notation

`'[_][_]'H.ProcessId,'M.ProcessId,'J1.ProcessId,'J2.ProcessId]`

means $H \mid M \mid J1 \mid J2$. The prefix notation makes it a little difficult to read the Maude solutions.

To summarize, CWB is useful for the CCS coordinating module, and it is not able to check the structures or computations behind the action names. On the other hand, Maude is not able to check the bisimulation of two CCS processes, and it does not support the hidden logic (yet). Consequently, it would be useful and beneficial to have an integrating software tool able to work as CWB and Maude extended to hidden logic [10].

6 Towards Temporal Specification and Model Checking

Since the semantics of the hiddenCCS specifications is given by labeled transition systems, we may use various temporal logics to express the properties of the specified systems. The atomic temporal propositions for the state sort h are equations having one of the forms:

$$q(-, d_1, \dots, d_n) = d \text{ with } q \text{ an attribute in } \Gamma \text{ and } d, d_1, \dots, d_n \in D(\mathcal{B}), \text{ or}$$

$\langle - \rangle = g(-, d_1, \dots, d_n)$ with g a method in Γ and $d_1, \dots, d_n \in D(\mathcal{B})$.

The intuitive meaning of an atomic Γ -proposition $q(-, d_1, \dots, d_n) = d$ is that we obtain d whenever we execute a query $q(-, d_1, \dots, d_n)$ over the current state. The intuitive meaning of $\langle - \rangle = g(-, d_1, \dots, d_n)$ is that there is a state which can be obtained from the current state by applying the method g over the current state with the arguments $d_1, \dots, d_n$.

The satisfaction relation of the atomic temporal propositions is defined as expected. Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$ with the state sort h , a $\mathcal{B}$ -model M , $st \in M_h$, and a CCS($\mathcal{B}$)-process P , then:

- (st, P) satisfies $q(-, d_1, \dots, d_n) = d$ iff $\llbracket q \rrbracket_M(st, d_1, \dots, d_n) = d$.
- (st, P) satisfies $\langle - \rangle = g(-, d_1, \dots, d_n)$ iff there are st' and P' such that $st' = \llbracket g \rrbracket_M(st, d_1, \dots, d_n)$ and $(st, P) \xrightarrow{\alpha} (st', P')$, where $\alpha = \text{synch}(g)$.

Remark 1. Given an object specification $\mathcal{B} = (\Sigma, \Gamma, E)$, a $\mathcal{B}$ -model M , and a method g in Γ with $\text{synch}(g) = \alpha$, then $(st, P) \models \langle - \rangle = g(-, d_1, \dots, d_n)$, iff $P \models \langle \alpha \rangle \mathbf{tt}$, for each state st in M , CCS($\mathcal{B}$)-process P , and $d_1, \dots, d_n \in D(\mathcal{B})$.

Computational Tree Logic (CTL) [2] is a branching time logic, meaning that its model of time is a tree-like structure in which the future is not determined; there are different paths in the future, any one of which might be the “actual” path that is desired. Using similar techniques to those presented in [9], it would be possible to build a canonical CTL model directly from the specification using a software tool. The temporal properties satisfied by any hiddenCCS model are the same as those satisfied by the canonical CTL model whenever some assumptions are satisfied. Once we have such a canonical CTL model, we can use an existing model checker such as SMV in order to check the desired temporal properties.

We can use the *Linear Temporal Logic* (LTL) model checker implemented in Maude [5]. LTL [2] is closely related to CTL in that it has similar expressive mechanisms, however, its formulas have meaning on computation paths, i.e., there is no explicit path quantifiers. For instance, an important property of a system specified in hiddenCCS is the consistency of the initial state and its associated coordinating process. This property can be checked as follows:

1. Considering a relaxed version of the hiddenCCS specification by removing the conditions regarding the satisfaction of the integration consistency requirements.
2. Expressing the consistency property by a formula of the form: for all paths, for all states, the resources are used only if they are available.
3. Model checking the above formula against a model provided by the relaxed specification.

For the JOBSHOP-CCS example, we remove the conditions `isAv(3* A)` (for hammer) and `isAv(4* A)` (for mallet) from the conditional part of the rewriting rules. Then we add to the agent specification the auxiliary attribute `isTired` that returns true iff the last method executed by the agent is `getHammer`, and

to the hammer specification an auxiliary operation `prev` that returns the previous state. Let `tired1` and `tired2` be the propositions that are true iff the corresponding agent “is tired”. Let `freeH` be the proposition expressing that the hammer is available in the previous state.

```
mod JOBSHOP-PREDS is
  protecting JOBSHOP-CCS .
  including SATISFACTION .
  subsort CcsJobshop < State .
  ops tired1 tired2 freeH : -> Prop .
  var J : Jobshop . var P : Term .
  eq < J, P > |= tired1 = isTired(1* J) .
  eq < J, P > |= tired2 = isTired(2* J) .
  eq < J, P > |= freeH = isAv(prev(3* J)) .
endm
```

The consistency property for the hammer is expressed by the temporal formula $\varphi \stackrel{\text{def}}{=} \Box((\text{tired1} \vee \text{tired2}) \implies \text{freeH})$. This formula can be checked by the LTL model checker implemented in Maude:

```
mod JOBSHOP-CHECK is
  including JOBSHOP-PREDS .
  including MODEL-CHECKER .
  including LTL-SIMPLIFIER .
  op initconf : -> CcsJobshop .
  op init : -> Jobshop .
  eq grade(1* init) = none .    eq grade(2* init) = none .
  eq isAv(3* init) = true .
  eq isTired(1* init) = false . eq isTired(2* init) = false .
  eq prev(3* init) = 3* init .
  eq initconf = < init, 'JobShop.ProcessId > .
endm
```

```
red modelCheck(initconf, [] ((tired1 \ / tired2) -> freeH)) .
```

and the result is:

```
Maude> red modelCheck(initconf, [] ((tired1 \ / tired2) -> freeH)) .
rewrites: 999079327 in 2348170ms cpu (2359850ms real) 425471 rewrites/sec
result Bool: true
```

The reader may note the effort (number of rewrites) made by Maude while checking this formula. On the other hand, if the initial state satisfies `isAv(3* init) = false`, the model checker produces a counterexample.

Remark 2. The LTL model checker works only if the set of reachable states is finite. The state space generated by the hiddenCCS specification JOBSHOP-CCS is infinite. It can be abstracted and reduced into a finite state space by considering the behavioral equivalence classes induced by the following equations:

```
eq endJob(A1) = 1* init .    eq endJob(A2) = 2* init .
eq release(H) = 3* init .    eq release(M) = 4* init .
```

The end of a job put the agent in a state behaviorally equivalent to its initial state, and the release of a resource put the resource in a state behaviorally equivalent to its initial state. The validity of these equations is shown using hidden logic in the same way as it is presented at the end of Section 2. The reduced system for an agent looks like in Figure 3.

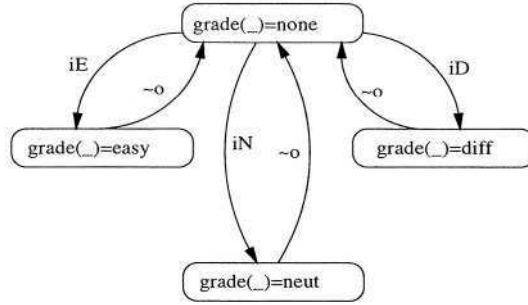


Fig. 3. The reduced system for AGENT-CCS

7 Conclusion and Related Work

This paper presents how a CCS process is used as a coordinating module that describes the global goal of a system of concurrent objects specified in hidden algebra. The integration of CCS and hidden algebra is given by appropriate semantic rules, producing a labeled transition system over configurations of form (hidden state, CCS process). We show some results regarding the strong bisimulation that allow to check the bisimulation in hiddenCCS by using tools for bisimulation in CCS and for behavioral equivalence in hidden algebra. Then we investigate how the existing tools CWB and Maude can be used and integrated to verify useful properties of the synchronizing concurrent objects. CWB is used to verify the strong bisimulation of two CCS processes, as well as to check properties expressed in propositional μ -calculus or CTL. The hiddenCCS specifications can be described in the rewriting logic of Maude, although we pay a price by losing the specific tools for verification of the behavioral properties. Since CCS, its Hennessy-Milner logic and LTL are implemented in Maude, we can use it as a software framework to represent the hiddenCCS specifications and their LTSs, as well as to verify some properties expressed in LTL.

The way we combine process algebra CCS for coordination aspects with hidden algebra for object descriptions allows to take advantage of both approaches: high abstraction level, expressiveness, and verification tools. Moreover, this approach offers new insights on integration consistency and global temporal properties.

We compare our approach with existing work for the integration of CCS with various formal specification languages. Proposals of integrating the algebraic specification language CASL with CCS have been made in [14]. The paper is mainly based on methodologies, and it does not present mathematical results regarding the model semantics. In this paper we present some results on the observational behavior of the whole system, considering the corresponding bisimulations in each component formalism together with a consistent semantic link between states and coordinating processes. Therefore we do not provide only a method of integrating the coordinating CCS with an algebraic speci-

cation language for concurrent objects, but some general results regarding the integrated bisimulation of the resulting system. Moreover, in contrast with [14], both coordinating CCS module and objects specifications are reusable.

Several works address the integration of model oriented specification languages as VDM, Z and B with CSP and CCS. We refer to those combining the Z notation and CCS for specifying concurrent systems [6,15]. Our approach has similarities with [15] by the fact that the operational semantics is given over configurations of form (state,process). The difference is that we use CCS for synchronization rather than as a value-passing formalism.

References

1. G. Ciobanu, D. Lucanu. Communicating Concurrent Objects in HiddenCCS. Accepted at WRLA '04. To appear in *Electronic Notes in Theoretical Computer Science*, Elsevier, 2004.
2. E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. MIT Press, 2000.
3. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada Maude: Specification and Programming in Rewriting Logic *Theoretical Computer Science*, vol. 285(2), pp. 187, 2002.
4. R. Cleaveland, J. Parrow, and B. Steffen. The Concurrency Workbench: a semantics-based tool for the verification of concurrent systems. In *ACM TOPLAS*, vol. 15(1), ACM Press, pp. 36-72, 1993.
5. S. Eker, J. Meseguer, and A. Sridharanarayanan The Maude LTL Model Checker In F.Gadducci, U.Montanari (Eds.) *4th WRLA, Electronic Notes in Theoretical Computer Science*, vol. 71, Elsevier, 2002.
6. A. J. Galloway, and W. Stoddart. An Operational Semantics for ZCCS. In M.G.Hinchey, Shaoying Liu (Eds.) *Proc. ICFEM.97*, pp. 272-282, IEEE Computer Society Press, 1997.
7. J. Goguen, and G. Malcolm. A hidden agenda. *Theoretical Computer Science* 245(1), pp.55-101, 2000.
8. J. Goguen, K. Lin, and G. Roşu. Circular Coinductive Rewriting. In *Proc. Automated Software Engineering '00*, IEEE Press, pp.123-131, 2000.
9. D. Lucanu, and G. Ciobanu. Model Checking for Object Specifications in Hidden Algebra, in *Proc. VMCAI, Lecture Notes in Computer Science* 2937, Springer, pp.97-109, 2004.
10. J. Meseguer, and G. Roşu Behavioral Membership Equational Logic. In *Proc. CMCS, Electronic Notes in Theoretical Computer Science*, vol. 65, Elsevier, 2002.
11. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
12. R. Milner. *Communicating and Mobile Systems: the π -calculus*. Cambridge University Press, 1999.
13. G. Roşu. *Hidden Logic*. PhD thesis, University of California at San Diego, 2000.
14. G. Salaün, M. Allemand, and C. Attiogbé. A Formalism Combining CCS and CASL. RESEARCH REPORT No 00.14, IRIN, Université de Nantes, 2001.
15. K. Taguchi, and K. Araki. The State-Based CCS Semantics for Concurrent Z Specification. In M.G.Hinchey, Shaoying Liu (Eds.) *Proc. ICFEM.97*, pp. 283-292, IEEE Computer Society Press, 1997.
16. A. Verdejo and N. Martí-Oliet. Implementing CCS in Maude 2. In F. Gadducci, U. Montanari (Eds.), *Proc. WRLA, Electronic Notes in Theoretical Computer Science*, vol. 71, pp. 239-257, Elsevier, 2002.

Understanding Object-Z Operations as Generalised Substitutions

Steve Dunne

School of Computing, University of Teesside
Middlesbrough, TS1 3BA, UK
`s.e.dunne@tees.ac.uk`

Abstract. Object-Z has a repertoire of operation operators and admits recursively defined operations to permit complex operations to be expressed compositionally via more primitive operation components. Although the operators are rigorously defined in the literature, some of these definitions are intuitively obscure. In this paper we interpret Object-Z class operations as generalised substitutions, thus investing them for the first time with a wp semantics. We can then bring to bear our theory of generalised substitutions to express Object-Z's operation operators in a new way which brings more intuitive clarity to their definitions. We also expose a flaw in the prevailing standard treatment of recursively defined operations in Object-Z, and draw on our theory of substitutions in proposing how to rectify that treatment.

1 Introduction

Object-Z [11] is a well-established extension of Z which greatly facilitates the specification of systems in an object-oriented style. It uses a Z-like schema style of specification of its class operations with a “before-after” relational semantics. Over the last ten years B [1] has also emerged as an industrial-strength formal development method with integrated tool support throughout the software development process from abstract specification through various intermediate stages of design to executable code generation. B uses a substitution style of specification of its abstract machine operations with a predicate-transformer semantics based on Dijkstra's wp [4]; operations are actually expressed in a programming-like abstract machine notation (AMN) and then translated by the tool into the language of generalised substitutions (GSL), inspired by Dijkstra's guarded command language but generalised to express not just executable programs but also more abstract non-executable specifications.

In this paper we interpret the operation operators of Object-Z in terms of B's generalised substitutions, thus endowing each operator with a formal wp semantics consistent with its existing relational definition in terms of Z schemas. By so doing we aim to illuminate the definitions of these operation operators, the better to understand certain aspects of those definitions which hitherto might have seemed obscure or arbitrary. Cavalcanti and Woodcock [2] have investigated and established a correspondence between Z's relational semantics of operations and

a wp semantics of those operations, but no-one appears to have done this for Object-Z before now. Moreover, we will use insight from our theory of generalised substitutions [5] to repair a significant flaw we uncover in the way recursive operations have so far been interpreted in Object-Z, thus securing a sound theoretical foundation for this important aspect of the language.

In the remainder of this section we introduce the various basic concepts and notations we employ in the rest of the paper. In Section 2 we summarise those aspects of our theory of generalised substitutions which are relevant here. In Section 3 we reach the essential subject of the paper, establishing the correspondence between Object-Z operations and their generalised-substitution counterparts, then expressing Object-Z's operation operators as substitution constructs. In Section 4 we explore how recursion is currently handled in Object-Z and expose the flaw we discover in that treatment, then show how by reference to the theory of generalised substitutions we can repair it.

Bunches. A bunch is essentially a flattened set. In contrast to set theory, bunch theory [7,8] makes no distinction between an element a and the singleton bunch comprising a . Rather than speaking of a bunch as *containing* its elements, we say it *comprises* or *consists of* them. If a and b are bunches then their *bunch union* a, b is the bunch that comprises the elements of both a and b . If a and b are bunches then their *bunch difference* $a \setminus b$ is the bunch that comprises those elements of a which are not also elements of b . If a and b are bunches then their *bunch intersection* $a \cap b$ is the bunch that comprises those elements of a which also belong to b . Among these bunch operators by convention bunch intersection “ $\cap$ ” has a higher precedence than bunch union “ $,$ ” which in turn has a higher precedence than bunch difference “ $\setminus$ ”. Thus, for example, the bunch expression $a \cap b, c \setminus d$ parses as $((a \cap b), c) \setminus d$.

If a and b are bunches we write $a : b$ to assert that a is *included* in b so that every element of a also belongs to b . Two bunches a and b are the same if they comprise the same elements: that is, $a : b \wedge b : a$. We abbreviate this as $a :: b$. We denote the *empty bunch* by $\emptyset$. Two bunches a and b are disjoint if $a \cap b : \emptyset$.

In this paper we frequently encounter bunches of variables. They provide a conveniently concise notation for marshalling otherwise unwieldy or indeterminate collections of variables. Thus we write $\forall w \bullet P$ where w is a metavariable denoting the bunch of basic quantified variables concerned in the quantification. When w is empty such a quantification degenerates just to P .

We will use systematic decoration for bunches and lists of variables so that, for example, if u is such a bunch or list then u' represents the corresponding bunch or list obtained by priming each constituent variable of u , while u_0 represents that obtained by zero-subscripting it.

If v and w are variable lists the predicate $v = w$ holds exactly if v and w are the same length and corresponding variables of the respective lists denote the same value. In the special case where v and w are empty lists $v = w$ is necessarily true, and $v \neq w$ necessarily false. We will sometimes slightly abuse our notation by writing a predicate such as $u = u'$ or an assignment like $u := u'$

even when u is merely a bunch rather than list of variables. To resolve these we assume an implicit lexical ordering over all variable names, so that where the context demands it a bunch of variables can be interpreted as the list obtained by arranging them in lexical order.

Universal implication. We use the symbol $\Rightarrow$ to denote universal implication over all variables in scope. Thus if w signifies this bunch of variables $P \Rightarrow Q$ means $\forall w \bullet P \Rightarrow Q$. We will write $P = Q$ when $P \Rightarrow Q$ and $Q \Rightarrow P$.

Syntactic substitution. If T is a predicate or generalised substitution, x a list of variables and E a corresponding list of expressions of appropriate types, then $T\langle E/x \rangle$ will denote the predicate or generalised substitution derived from T by replacing all its free occurrences of any constituent of x by the corresponding constituent of E . Incidentally, if x and E are empty lists then $T\langle E/x \rangle$ is T .

Alphabetised relations. Hoare and He [9] introduce the concept of an *alphabetised relation* as a pair (α, P) where α is a bunch of variables and P is a predicate over those variables: that is, P contains no free variables other than from α . Where the context unambiguously determines its alphabet we often refer to a relation by its predicate alone. A *binary relation* is a pair (α, P) where $\alpha :: in, out'$; here in is a set of unprimed input variables and out' a set of primed output variables, and P is a predicate containing no free variables other than from α . A *homogeneous binary relation* is a binary relation whose output alphabet out' is obtained from the input alphabet in simply by priming each of the latter's variables, so that $in :: out$. A *condition* is a homogeneous binary relation whose predicate contains no free primed variables of its alphabet.

2 A Theory of Generalised Substitutions

In [5] we define a generalised substitution language slightly extended from Abrial's original one in [1], and then derive from just three fundamental healthiness properties all the characteristic properties of substitutions on which B relies, as well as certain others. Generalised substitutions are interpreted in the context of a bunch of typed variables characterising the program state. In our formulation a generalised substitution S is characterised by its active frame s which is the bunch of state variables to which it can assign values, and its wp predicate-transformer effect $[S]$ which acts on postconditions over the state. Our basic substitutions and substitution operators are defined in Table 1, in which S and T represent typical generalised substitutions with respective frames s and t which may overlap, and P and Q are conditions on the state.

Characteristic predicates. Several important characteristic predicates can be extracted from a generalised substitution. Three of them are defined in Table 2. They can be interpreted as homogeneous binary relations in the context of

Table 1. Basic generalised substitutions and operators

name	syntax	frame	wp(sub, Q)	notes
skip	skip	$\emptyset$	Q	
assignment	$u := E$	u	$Q\langle E/u \rangle$	1
precondition	$P \mid S$	s	$P \wedge [S] Q$	
guard	$P \Longrightarrow S$	s	$P \Rightarrow [S] Q$	
frame extension	S_z	s, z	$[S] Q$	2
bounded choice	$S \parallel T$	s, t	$[S] Q \wedge [T] Q$	
unbounded choice	$@ z . S$	$s \setminus z$	$\forall z \bullet [S] Q$	3
sequential composition	$S ; T$	s, t	$[S] [T] Q$	

- 1 u is a list of distinct variables and E is a corresponding list of appropriately typed values.
- 2 z is a bunch of variables not necessarily disjoint from s .
- 3 z is a bunch of variables which must be fresh with respect to the current state and therefore non-free in Q . They locally enrich the state in which S is interpreted.

the alphabet v, v' , where v is the bunch of variables characterising the state: $\text{trm}(S)$ is S 's implicit precondition and $\text{fis}(S)$ its implicit guard, while $\text{prd}(S)$ is the before-after relation which expresses the effect on the state of executing S by relating unprimed before-states to primed corresponding after-states.

An important result derived in [5] is that any generalised substitution S with frame s can be expressed in the following normal form

$$\text{trm}(S) \mid @ s' . \text{prd}(S) \Longrightarrow s := s'$$

showing that S is completely characterised by its frame, its trm and its prd . This is used to define indirectly the two further substitution constructs seen in Table 3. The indeterminate assignment $u : P$ can be informally interpreted as “set the variables of u to any values which satisfy P ”, where any zero-subscripted variables of u_0 appearing in P represent before-values of those state variables. The parallel composition $S \parallel T$ represents simultaneous execution of S and T ; we note that Abrial's original formulation of this construct in [1] restricts it to

Table 2. Some characteristic predicates of a substitution S with frame s

name	syntax	definition
termination	$\text{trm}(S)$	$[S] \text{ true}$
feasibility	$\text{fis}(S)$	$\neg [S] \text{ false}$
before-after	$\text{prd}(S)$	$\neg [S] s \neq s'$

Table 3. Two further indirectly-defined substitution constructs

sub	syntax	frame	trm	prd
indeterminate assignment	$u : P$	u	true	$P\langle u, u' / u_0, u \rangle$
parallel composition	$S \parallel T$	s, t	$\text{trm}(S) \wedge \text{trm}(T)$	$(\text{trm}(T) \Rightarrow \text{prd}(S)) \wedge (\text{trm}(S) \Rightarrow \text{prd}(T))$

cases where the frames of S and T are disjoint, whereas we dispense with this restriction.

Operations in B. In this paper we abstract the specification of a B operation to be a triple (S, i, o) where i is a bunch of inputs, o a bunch of outputs and S a generalised substitution expressing the operation's effect. For such a specification to be well-formed $\text{frame}(S)$ must be disjoint from i and must include o , while S must make no reference to the before-value of any variable of o .

The refinement lattice of substitutions. If S and T are two substitutions on a bunch v of state variables with respective frames s and t then S is refined by T , written $S \sqsubseteq T$, iff $s : t$ and $[S] Q \Rightarrow [T] Q$ for every postcondition Q over v . All the constructors of our generalised substitution language are monotonic with respect to this ordering. Generalised substitutions form a complete lattice under it, whose top and bottom are respectively $\text{false} \Rightarrow v := v$ and $\text{false} \mid \text{skip}$. Since all the constructors of the GSL are monotonic with respect to $\sqsubseteq$, the well-known Knaster-Tarski theorem[13] ensures that any GSL expression $F(S)$ in S has both a least and a greatest fixed point with respect to this ordering. Our theory of generalised substitutions therefore follows the conventional computing approach of interpreting a recursive definition of the form $S =_{df} F(S)$ as defining S to be the least fixed point of $F(S)$ with respect to our refinement ordering $\sqsubseteq$.

Substitutions with the same frame u form a sublattice whose top and bottom are respectively $\text{false} \Rightarrow u := u$ and $\text{false} \mid u := u$.

Always-terminating substitutions with the same frame u form a sublattice whose top and bottom are respectively $\text{false} \Rightarrow u := u$ and $u : u \in T$, where T is the appropriate product of types for u , which we can interpret as “assign any values to the variables of u consistent with their types”.

3 Object-Z Operations

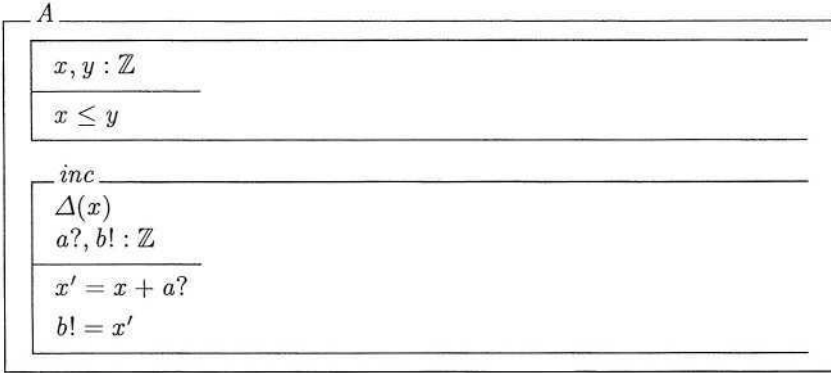
Object-Z class operations have a *blocking* semantics rather than the diverging one more usually associated with operations in Z. They can only occur when their preconditions are satisfied, otherwise they are blocked – that is, not available to the application. Unlike a normal Z operation, which diverges (behaves chaotically, perhaps abortively) when invoked outside its precondition, an Object-Z operation therefore never diverges. It thus corresponds to a generalised substitution whose frame comprises the delta list and outputs of the operation, whose trm is always true and whose prd reflects its operation schema property. The fis of this substitution reflects the schema precondition of the operation, characterising where the operation is enabled (*i.e.* not blocked).

The general form of an Object-Z class operation schema is

Op
$\Delta(u)$
auxiliary declarations
P

where the delta list u is a bunch of state variables of the class which the operation may change, the auxiliary declarations usually introduce $?$ - and $!$ -decorated variables, whose base-names are conventionally distinct, representing respectively the inputs and outputs of the operation, and P is a predicate constraining these and the unprimed (before) and primed (after) state variables of the class. For convenience we will assume that Op is fully normalized so that the predicate P expresses its entire property, including the class’s state invariant implicitly imposed both on the unprimed (before) and the primed (after) class variables.

Let w be the bunch of all state variables of the class, i the bunch of basenames of the $?$ -decorated input variables of Op , and o the bunch of basenames of the $!$ -decorated output variables of Op . Now let P_0 be the predicate derived from P by simultaneously “undecorating” any $?$ - or $!$ -decorated variable, “unpriming” any primed variable of w' and zero-subscripting any unprimed variable of u which appears in P : that is to say, P_0 is $P(i, o, w, u_0 / i?, o!, w', u)$. Then the indeterminate assignment $u, o : P_0$ expresses the meaning of Op . Recall we can informally interpret this as “set the variables of u and o to any values which satisfy P_0 , where any zero-subscripted variable of u_0 appearing in P_0 denotes its before-value”. Thus, for example, in the following Object-Z class



the fully expanded property P of operation *inc* is

$$x : \mathbb{Z} \wedge y : \mathbb{Z} \wedge x \leq y \wedge x' : \mathbb{Z} \wedge y' : \mathbb{Z} \wedge x' \leq y' \wedge \\ a? : \mathbb{Z} \wedge b! : \mathbb{Z} \wedge x' = x + a? \wedge b! = x'$$

and the corresponding modified property P_0 is¹

$$x_0 : \mathbb{Z} \wedge y : \mathbb{Z} \wedge x_0 \leq y \wedge x : \mathbb{Z} \wedge x \leq y \wedge \\ a : \mathbb{Z} \wedge b : \mathbb{Z} \wedge x = x_0 + a \wedge b = x$$

The meaning of operation *inc* is then expressed by the indeterminate assignment $x, b : P_0$, which is equivalent to the guarded assignment

$$x : \mathbb{Z} \wedge y : \mathbb{Z} \wedge a : \mathbb{Z} \wedge x \leq y \wedge x + a \leq y \implies x, b := x + a, x + a$$

3.1 Object-Z Operation Operators

Object-Z provides a repertoire of operation operators similar to the schema operators of Z but for use specifically in combining operations, so that a new operation may be expressed as a combination of simpler subsidiary component operations. We use identifiers A and B in what follows to stand for typical operations. We slightly overload these identifiers also to denote the generalised substitutions which express these same operations. We denote their bunches of input variables by $\text{in}(A)$ and $\text{in}(B)$ and their bunches of output variables by $\text{out}(A)$ and $\text{out}(B)$. We discard the Z-style ?- and !-decorations from these input and output variables, using only their undecorated base-names which we assume are all distinct within an operation.

Except where otherwise indicated in the operators below, the input of the combined operation is the union of the inputs of the two constituent operations, and its output is the union of their outputs.

¹ Notice that of A 's two state variables x, y in P only x , being in *inc*'s delta list, is replaced in P_0 by x_0 , while y , being outside the delta list, remains as y .

- The **conjunction** $A \wedge B$ models the simultaneous occurrence of operations A and B . As a generalised substitution it is expressed by the parallel composition $A \parallel B$, since as $\text{trm}(A)$ and $\text{trm}(B)$ are identically true we have

$$\text{prd}(A \parallel B) = \text{prd}(A) \wedge \text{prd}(B)$$

- The **nondeterministic choice** $A \sqcup B$ models internal or demonic² choice between operations. It is only defined for operations with the same auxiliary variables, and therefore in particular the same inputs and outputs. As a generalised substitution it is expressed by the bounded choice $A \sqcup B$. Recall from Table 1 that the GSL's $\sqcup$ is defined in wp terms simply by logical conjunction. Moreover, in our theory of generalised substitutions it is easy to show that

$$\text{prd}(A \sqcup B) = \text{prd}(A) \wedge u' = u \vee \text{prd}(B) \wedge v' = v$$

where $u :: \text{frame}(B) \setminus \text{frame}(A)$ and $v :: \text{frame}(A) \setminus \text{frame}(B)$, which illuminates and vindicates the relatively more opaque Object-Z definition of this operator in [11].

- The (non-associative) **parallel composition** $A \parallel B$ models communication between simultaneously occurring operations, with the inputs of either operation being identified with any output of the other operation with the same basename, these unified inputs and outputs being hidden in the resulting operation. As a generalised substitution it is expressed by $\text{Q} z . (A \parallel B \parallel z := z)$, where z is the bunch of all so-identified inputs and outputs: that is to say, $z :: \text{in}(A) \cap \text{out}(B), \text{in}(B) \cap \text{out}(A)$. The variables of z are hidden from both the inputs and outputs of the composition, so $\text{in}(A \parallel B) :: \text{in}(A), \text{in}(B) \setminus z$ and $\text{out}(A \parallel B) :: \text{out}(A), \text{out}(B) \setminus z$. The assignment $z := z$ in the GSL parallel composition coerces the (after) value of each identified output emanating from its issuing operation to be the same as the (before) value of the corresponding input into its receiving operation.
- The **associative parallel composition** $A \parallel\!\!\parallel B$ also models communication between simultaneously occurring operations, again with the inputs of either operation being identified with any output of the other operation with the same basename, but in this case the identified input/outputs remain visible as outputs of the combined operation. To express this as a generalised substitution we again define z as the bunch of all so-identified inputs and outputs: that is to say, $z :: \text{in}(A) \cap \text{out}(B), \text{in}(B) \cap \text{out}(A)$. We cannot quantify over z in this case since it must remain visible as part of the output

² Since it is exercised, so we imagine, by a demon who inhabits the machine on which our system will execute. The demon, however, abhors a miracle: faced with a choice between feasible (enabled) and infeasible (blocked) operations, he is obliged to choose the former; such choices are sometimes misdescribed, ironically, as angelic.

of the composition, so we introduce z° as a systematically decorated variant of z and $A^\circ = A\langle z^\circ/z \rangle$ and $B^\circ = B\langle z^\circ/z \rangle$ as corresponding variants of A and B . The composition is then expressed as

$$\mathbb{Q} z^\circ . (A^\circ \parallel B^\circ \parallel z^\circ := z^\circ \parallel z := z^\circ)$$

Here the assignment $z^\circ := z^\circ$ plays the same role as the $z := z$ in the non-associative composition. The assignment $z := z^\circ$ captures these values as visible outputs in z . The variables of z are hidden from the inputs but not the outputs of the composition, so

$$\text{in}(A \parallel B) :: \text{in}(A), \text{in}(B) \setminus z \quad \text{and} \quad \text{out}(A \parallel B) :: \text{out}(A), \text{out}(B)$$

- The **sequential composition** $A \circ B$ models two operations occurring in sequence. It is only defined when all inputs of the first operation A are disjoint from the outputs of the second operation B . Outputs of A are identified with inputs of B with the same basename, these identified inputs and outputs being hidden in the resulting operation. As a generalised substitution it is expressed by $\mathbb{Q} z . (A ; B)$ where z is the bunch of outputs of A identified with inputs of B : that is to say, $z :: \text{out}(A) \cap \text{in}(B)$. The variables of z are hidden from both the inputs and outputs of the composition, so

$$\text{in}(A \circ B) :: \text{in}(A), \text{in}(B) \setminus z \quad \text{and} \quad \text{out}(A \circ B) :: \text{out}(A), \text{out}(B) \setminus z$$

- The **scope enrichment** $[u \mid P] \bullet A$ brings new declarations u constrained by condition P into the scope of operation A so as to provide a locally enriched environment in which to interpret it.³ As a generalised substitution it is expressed by $\mathbb{Q} u . P \implies A$. The variables u must not already be declared in A . The inputs and outputs such a scope-enriched operation are just those of the operation concerned, so

$$\text{in}([u \mid P] \bullet A) :: \text{in}(A) \quad \text{and} \quad \text{out}([u \mid P] \bullet A) :: \text{out}(A)$$

- The **hiding** $A \setminus (u)$ hides the auxiliary variables u , which may include inputs and outputs, in operation A . As a generalised substitution it is expressed by $\mathbb{Q} u . A$. Any inputs or outputs of A occurring in u are hidden, so

$$\text{in}(A \setminus (u)) :: \text{in}(A) \setminus u \quad \text{and} \quad \text{out}(A \setminus (u)) :: \text{out}(A) \setminus u$$

We summarise all these operators in Table 4.

4 Recursively-Defined Operations

Smith [11, section 3.7.5] discusses recursive operation definition in Object-Z, concluding with a brief reference to fixed-point theory and characterisation of

³ Our unary scope-enrichment operator here is actually a specialised version, sufficient for most practical specification purposes, of the binary one defined in [11].

Table 4. Object-Z operation operators and their GSL interpretations

Object-Z	GSL	inputs	outputs	notes
$A \wedge B$	$A \parallel B$	$\text{in}(A), \text{in}(B)$	$\text{out}(A), \text{out}(B)$	
$A \parallel B$	$A \parallel B$	$\text{in}(A)$	$\text{out}(A)$	1
$A \parallel B$	$\mathbb{Q} z. (A \parallel B \parallel z := z)$	$\text{in}(A), \text{in}(B) \setminus z$	$\text{out}(A), \text{out}(B) \setminus z$	2
$A \parallel B$	$\mathbb{Q} z^\circ. (A^\circ \parallel B^\circ \parallel z, z^\circ := z^\circ, z^\circ)$	$\text{in}(A), \text{in}(B) \setminus z$	$\text{out}(A), \text{out}(B)$	2, 3, 4
$A \circ B$	$\mathbb{Q} z. (A ; B)$	$\text{in}(A), \text{in}(B) \setminus z$	$\text{out}(A), \text{out}(B) \setminus z$	5
$[u \mid P] \bullet A$	$\mathbb{Q} u. P \implies A$	$\text{in}(A)$	$\text{out}(A)$	6
$A \setminus (u)$	$\mathbb{Q} u. A$	$\text{in}(A) \setminus u$	$\text{out}(A) \setminus u$	

- 1 A and B must have same inputs and outputs
- 2 $z :: \text{in}(A) \cap \text{out}(B), \text{in}(B) \cap \text{out}(A)$
- 3 z° is a systematically decorated variant of z
- 4 A° is $A\langle z^\circ/z \rangle$ and B° is $B\langle z^\circ/z \rangle$
- 5 $\text{in}(A) \cap \text{out}(B) : \emptyset$ and $z :: \text{out}(A) \cap \text{in}(B)$
- 6 no variable of u must be declared in A

an ordering of operations on which he suggests such a fixed-point treatment of recursion in Object-Z can be based. But on close inspection this ordering proves to be deficient for such a purpose since it isn't anti-symmetric. He must have recognised this deficiency, since in [12] he gives a more comprehensive treatment of the same subject based on a different ordering $\sqsubseteq_{Op}$ on operations such that

$$\begin{aligned}
 & [\Delta(u) \ d_1 \mid p] \sqsubseteq_{Op} [\Delta(v) \ d_2 \mid q] \\
 & \Leftrightarrow \\
 & u : v \ \wedge \ d_1 : d_2 \ \wedge \ (p \Rightarrow q)
 \end{aligned}$$

Unfortunately, this ordering turns out to be deficient too, since –contrary to what is claimed in [12]– it is not the case that every operation operator of Object-Z is monotonic in each of its arguments with respect to $\sqsubseteq_{Op}$. We demonstrate this by the following counterexample which shows that nondeterministic choice $[]$ isn't monotonic with respect to $\sqsubseteq_{Op}$. Let

$$\begin{aligned}
 Op1 & \hat{=} [\text{false}] \\
 Op2 & \hat{=} [\Delta(x) \mid \text{false}] \\
 Op & \hat{=} [\text{true}]
 \end{aligned}$$

Then clearly $Op1 \sqsubseteq_{Op} Op2$, but

$$Op1 \sqcup Op = [\text{false} \vee \text{true}] = [\text{true}]$$

while

$$Op2 \sqcup Op = [\Delta(x) \mid \text{false} \vee (\text{true} \wedge x = x')] = [\Delta(x) \mid x = x']$$

Hence it isn't the case that

$$Op1 \sqcup Op \sqsubseteq_{Op} Op2 \sqcup Op$$

since $\text{true} \Rightarrow x = x'$ is false. Nor can the problem be rectified by the simple expedient of reversing the direction of the implication in the definition of $\sqsubseteq_{Op}$. To see this let us denote such a variant of $\sqsubseteq_{Op}$ by $\sqsubseteq^{Op}$, so that

$$\begin{aligned} [\Delta(u) \ d_1 \mid p] &\sqsubseteq^{Op} [\Delta(v) \ d_2 \mid q] \\ \Leftrightarrow \\ u : v \ \wedge \ d_1 : d_2 \ \wedge \ (q \Rightarrow p) \end{aligned}$$

and let

$$\begin{aligned} Op1 &\hat{=} [\text{true}] \\ Op2 &\hat{=} [\Delta(x) \mid \text{true}] \\ Op &\hat{=} [\Delta(x) \mid \text{false}] \end{aligned}$$

Then clearly $Op1 \sqsubseteq^{Op} Op2$, but

$$Op1 \sqcup Op = [\Delta(x) \mid (\text{true} \wedge x = x') \vee \text{false}] = [\Delta(x) \mid x = x']$$

while

$$Op2 \sqcup Op = [\Delta(x) \mid \text{true} \vee \text{false}] = [\Delta(x) \mid \text{true}]$$

Hence it isn't the case that

$$Op1 \sqcup Op \sqsubseteq^{Op} Op2 \sqcup Op$$

since again $\text{true} \Rightarrow x = x'$ is false. So $\sqcup$ isn't monotonic with respect to $\sqsubseteq^{Op}$ either. The ramifications of this are serious: it ostensibly invalidates the whole theoretical basis of Smith's treatment of recursive operations in Object-Z; yet many Object-Z users would regard them as an indispensable feature of the language.

4.1 More about Monotonicity

The lack of monotonicity of some schema operators of Z is well known. It is a recurring theme in the Z literature, having for example recently been extensively

explored by Groves [6] and by Deutsch *et al* [3]. It is important to stress that the monotonicity of concern to these authors is that of the Z schema operators with respect to operation refinement. We recall at this point that operation refinement is a relationship between operations over the same state space with the same inputs and outputs. This lack of monotonicity arises from what Derrick and Boiten [10] call the *contract* interpretation of a Z operation, by which within its precondition the operation is guaranteed to deliver a well-defined result but outside of which, although it may still be applied, nothing at all can be guaranteed. They also offer an alternative *behavioural* interpretation, in which the operation is simply blocked (incapable of being applied) outside its precondition (more properly in this interpretation called its *guard*). It seems a curiously little-regarded fact among Z users that in the context of such a behavioural interpretation all the schema operators become monotonic with respect to refinement.

Since Object-Z operations already have a behavioural semantics the operation operators avoid the refinement monotonicity problems which beset the Z schema operators, in the sense that when the application of the operators is confined to operations with the same delta list and declarations, for which refinement is characterised exactly as for Z operations under the behavioural interpretation, the operators are indeed all monotonic with respect to refinement.

In [12] Smith sought a single comprehensive complete partial ordering over *all* the possible operations of a class (with whatever delta lists and declarations), on which to base his fixed-point treatment of recursive operations. But as we have shown, both the $\sqsubseteq_{Op}$ ordering he actually formulated and our speculative $\sqsubseteq^{Op}$ variant of it are unsuitable for such a purpose since at least one of the operation operators, $[]$, is non-monotonic with respect to each of them.

4.2 An Alternative Approach

The Object-Z operation operators do have two obvious monotonicity properties with respect to bunch inclusion, concerning the delta lists and auxiliary declarations of Object-Z operations. Let $F(X)$ be a parameterised operation expression in operation parameter X , and let A and B be the operations $[\Delta(u) \ d_1 \mid p]$ and $[\Delta(v) \ d_2 \mid q]$ respectively. Then

1. if $u : v$ then $\text{delta\_list}(F(A)) : \text{delta\_list}(F(B))$
2. if $d_1 : d_2$ then $\text{declarations}(F(A)) : \text{declarations}(F(B))$

We can capitalise on these properties to define the delta list and declarations of a recursively defined operation $Op \hat{=} F(Op)$ as follows:

$$\begin{aligned} \text{delta\_list}(Op) &=_{df} \text{delta\_list}(F([\text{true}])) \\ \text{declarations}(Op) &=_{df} \text{declarations}(F([\text{true}])) \end{aligned}$$

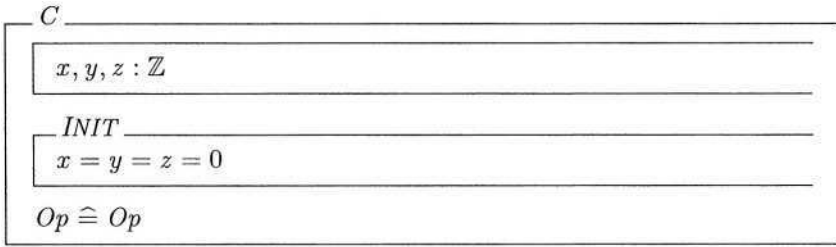
the significance of $[\text{true}]$ here, of course, being that it is an operation with an empty delta list and no declarations. Having in this way established the delta list and declarations of our recursive operation Op , let us denote them by u and d

respectively. We now consider all possible operations of the class with this delta list u and these declarations d . Under their refinement ordering $\sqsubseteq_{u,d}$, where

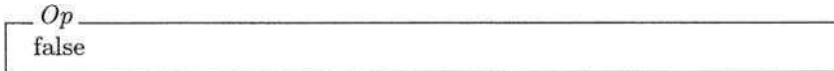
$$[\Delta(u) \ d \mid p] \sqsubseteq_{u,d} [\Delta(u) \ d \mid q] \quad \Leftrightarrow \quad q \Rightarrow p$$

such operations form a complete lattice which we will call $OP_{u,d}$. The set of generalised substitutions expressing these Object-Z operations forms a corresponding lattice under refinement. Our theory of generalised substitutions ensures that all the GSL's substitution constructs are monotonic with respect to refinement. Since we have expressed all the Object-Z operation operators in terms of such substitution constructs we can infer that these operators are monotonic on our lattice $OP_{u,d}$. We could therefore take Op to be either of the two distinguished fixed points of F in this lattice: the least or the greatest; for consistency with Smith's approach we will take the greatest fixed point.

To take an extreme example, consider the operation Op defined in the following class C :



In our treatment, as in Smith's, Op is equivalent to



So in fact Op is always blocked and can never be successfully invoked as an operation in its own right.

5 Conclusion

We have exhibited a precise correspondence between the operations of an Object-Z class and those of a B abstract machine. To do so we had to interpret each of Object-Z's operation operators in terms of the substitution constructs of B's GSL. This illuminated some of the more obscure aspects of those operators' characterisations in [11]. The correspondence we have exhibited effectively establishes for the first time a formal wp-based semantics for Object-Z operations which provides a reliable basis for addressing, for example, such thorny little questions as whether, in the context of a class with state variable x , the empty delta-list operations $A \hat{=} [true]$ and $B \hat{=} [x = x']$ can be considered synonymous.

We have exposed a flaw in the standard fixed-point treatment of recursive operations in Object-Z as represented in [12], and have then looked to the corresponding treatment of recursive substitutions in our theory of generalised substitutions for inspiration as to how to rectify this.

More speculatively, our work prompts the interesting question: might a more thorough-going integration of Object-Z and B be desirable in the long term? The existing Object-Z framework would of course be maintained for expressing the static structure, including class composition and inheritance hierarchies, of an object-oriented system's specification, but its dynamics (i.e. class operations) would be expressed directly as generalised substitutions. Apart from giving specifiers a more completely expressive notation with both preconditions and guards to express termination as well as feasibility conditions for operations, a substitution notation is often more natural and intuitive than a Z-style before-after relational one for expressing operational dynamics. The introduction of diverging operations into the Object-Z landscape would allow a more operationally-intuitive interpretation of pathological recursions such as our $Op \hat{=} Op$, which the practical programmer is more inclined to perceive as a non-terminating recursion than a blocked computation.

In a B abstract machine the interrelationship between the state invariant and the operation specifications is quite different from that in an Object-Z class. In B the operations are written independently of the invariant. The operation's author then incurs the obligation of proving that his operation will maintain the invariant. This provides a useful level of consistency-checking of B specifications which has no counterpart in Object-Z because there the class invariant is implicitly subsumed in every operation specification. The GSL's wp-based semantics seems particularly suited for mechanised reasoning both about specification consistency and refinement, as witnessed by the successful implementation of such reasoning in B's support tools.

Acknowledgements. Thanks are due to Graeme Smith for his encouragement to write this paper and his tireless responses to our queries about Object-Z which that precipitated, and to our shepherd Eerke Boiten for his pertinent suggestions for improving our original draft.

References

1. J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
2. A. Cavalcanti and J. Woodcock. A weakest-precondition semantics for Z. *The Computer Journal*, 41(1):663–675, 1998.
3. M. Deutsch, M.C. Henson, and S. Reeves. Operation refinement and monotonicity in the schema calculus. In D. Bert, J.P. Bowen, S. King, and M. Walden, editors, *ZB2003: Formal Specification and Development in Z and B*, number 2651 in Lecture Notes in Computer Science, pages 103–126. Springer-Verlag, 2003.
4. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

5. S.E. Dunne. A theory of generalised substitutions. In D. Bert, J.P. Bowen, M.C. Henson, and K. Robinson, editors, *ZB2002: Formal Specification and Development in Z and B*, number 2272 in Lecture Notes in Computer Science, pages 270–290. Springer-Verlag, 2002.
6. L. Groves. Refinement and the Z schema calculus. In J. Derrick, E. Boiten, J. Woodcock, and J. von Wright, editors, *REFINE '02, an FME-sponsored Refinement Workshop in collaboration with BCS FACS, Copenhagen*, Electronic Notes in Theoretical Computer Science. Elsevier, July 2002.
<http://www.elsevier.nl/locate/entcs>.
7. E.C.R. Hehner. Bunch theory: a simple set theory for computer science. *Information Processing Letters*, 12(1):26–30, 1981.
8. E.C.R. Hehner. *A Practical Theory of Programming*. Springer-Verlag, 1993.
9. C.A.R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall, 1998.
10. Derrick J. and Boiten E. *Refinement in Z and Object-Z*. Springer, 2001.
11. G. Smith. *The Object-Z Specification Language*. Advances in Formal Methods. Kluwer Academic Publishers, 2000.
12. G. Smith. Recursive schema definitions in Object-Z. In J.P. Bowen, S.E. Dunne, A.J. Galloway, and S. King, editors, *ZB2000: Formal Specification and Development in B and Z*, number 1878 in Lecture Notes in Computer Science, pages 42–58. Springer, 2000.
13. A. Tarski. A lattice-theoretical fixed-point theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

Embeddings of Hybrid Automata in Process Algebra

Tim A.C. Willemse

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O.Box 513, 5600 MB Eindhoven, The Netherlands
tel: +31 40 2475004, fax: +31 40 2475361, T.A.C.Willemse@TUE.nl

Abstract. We study the expressive power of two modelling formalisms, viz. *hybrid automata* and μCRL_t . The automaton based language of hybrid automata is a popular formalism that is used for describing and analysing the behaviours of hybrid systems. The process algebraic language μCRL_t is designed for specifying real-time and data-dependent systems and to reason about such systems. We show that every hybrid automaton can be translated to a μCRL_t expression without loss of information, i.e. the translation is equivalence preserving. This proves that μCRL_t is at least as expressive as the modelling language of hybrid automata. Subsequently, we extend the standard model of a hybrid automaton to deal with communications via shared continuous variables. We show that the resulting *enhanced hybrid automata* can also be embedded in μCRL_t .

Keywords: Hybrid Systems, Real-Time Systems, Hybrid Automata, Process Algebra, μCRL_t , Expressive Power

1 Introduction

A hybrid system is a system in which the essence of its behaviour cannot be captured by describing only its continuous components or its discrete components; the combination of both is essential for a correct representation of the system. The interest in such systems has grown enormously in the last decades. Application areas for hybrid systems are abundant, ranging from process control to avionics.

Languages that are tailored for describing models of hybrid systems have been proposed in the past, some of which have become quite popular. Many of these languages come with their own set of analysis tools, e.g. the χ -language [3] and its simulator, and hybrid automata [1,9] and the model checker HyTech [10]. Much of the (computer science initiated) research seems to concentrate on the latter language.

This paper focuses on the expressive power of *hybrid automata* in relation to a standard real-time process algebra, called μCRL_t [6,7]. We show that we can translate, or *embed*, all hybrid automata to μCRL_t expressions. This translation is such that the strongest, natural equivalence of hybrid automata (timed

bisimulation) is preserved and reflected. We furthermore consider an enhancement of standard hybrid automata. These *enhanced hybrid automata* have basic capabilities for communicating via shared continuous variables. This type of communication is commonly used in disciplines, other than computer science. We show that these enhanced hybrid automata also can be faithfully embedded in μCRL_t .

The fact that all (enhanced) hybrid automata can be embedded in μCRL_t shows that the latter is indeed at least as expressive as (enhanced) hybrid automata. In fact, we also show that there are simple μCRL_t expressions that do not have a counterpart (enhanced) hybrid automaton, making μCRL_t strictly more expressive than (enhanced) hybrid automata.

The results are important, as they can be used for deciding which language to use in a particular situation. Moreover, both embeddings lay the groundwork for examining which process algebraic techniques can be imported and brought to bear in the verification of hybrid systems.

This paper is organised as follows. Section 2 briefly introduces the language μCRL_t , and in Section 3 we present the model of hybrid automata and show that these can be interpreted in μCRL_t . Then, in Section 4, we extend the classical hybrid automaton model to allow for communications via shared continuous variables. We show that these can also be translated to μCRL_t expressions. We finish with closing remarks in Section 5.

2 An Overview of μCRL_t

In this section, we provide a brief, informal introduction to the constructs of the language μCRL_t —pronounce *timed mu-C-R-L*—that are relevant to this paper. A formal treatment of the operational semantics can be found in Appendix A. For a more detailed overview of the language, its complete axiom system and several meta results, we refer to [5,11,12].

Data is an integral part of the language μCRL_t and its untimed predecessor μCRL [6,7]. For the exhibition of the remainder of the theory, we assume we work in the context of a data algebra without explicitly mentioning its constituent components. As a convention, we assume the data algebra contains all the required data types; in particular, we always have the time-domain $\mathbb{R}_{\geq 0}$, with the usual ordering $\leq$ and least element $0: \rightarrow \mathbb{R}_{\geq 0}$.

Processes from a set $\mathbb{P}$ are the main objects in the language μCRL_t . A set of actions Act is assumed (actions are functions from a data domain to the set of processes). An action $a \in \text{Act}$ represents an atomic event. The process representing no behaviour, i.e. the process that cannot perform any actions is denoted δ . This constant is often referred to as *deadlock* or *inaction*. All actions terminate successfully immediately after execution; δ cannot be executed.

Processes are constructed using several operators. The main operators are alternative composition ($p + q$ for processes p and q , modelling a non-deterministic choice between processes p and q) and sequential composition ($p \cdot q$ for processes p and q , modelling the process that behaves as p , and upon termination of p , continues to behave as process q). Time is introduced using a binary time-stamping

operator $p \dot{c} t$, representing the behaviour of process p that starts exactly at absolute point of time t . Note that the process $\delta \dot{c} t$ denotes the process that can idle up to time t , but never executes an action, in effect modelling a *time-lock* at time t . The neutral element for alternative composition is $\delta \dot{c} 0$.

Conditional behaviour is denoted using a ternary operator (we write $p \triangleleft b \triangleright q$ when we mean process p if b holds and else process q). The process $[b]::\rightarrow p$ serves as a shorthand for the process $p \triangleleft b \triangleright \delta \dot{c} 0$, which represents the process p under the premise that b holds. Recursive behaviour is specified using equations.

The parallel composition of processes is denoted $p \parallel q$ for processes p and q . It models the interleavings of p , q , and their synchronisations. The outcome of synchronisation is prescribed by a separate, *partial* communication function γ , defining the result of a communication of two actions (e.g. $\gamma(a, b) = c$ denotes the communication between actions a and b , resulting in action c). Two parameterised actions $a(n)$ and $b(n')$ can communicate to action $c(n'')$ only if the communication between actions a and b is defined and results in action c (i.e. $\gamma(a, b) = c$) and their data parameters agree (i.e. $n'' = n' = n$).

The communication function γ is used to specify *when* communication is possible; this, however, does not mean communication is enforced. To this end, we allow for the encapsulation of actions, using the operator $\partial_H(p)$, where H is the set of all actions to be blocked in p . Renaming of actions, for instance to protect them from encapsulation, is achieved by the operator $\rho_R(p)$, where R is a relabelling. We use the following convention: rather than writing $\rho_R(\partial_H(p))$, we write $\rho_R \circ \partial_H(p)$.

Data-dependent alternative quantification is written $\sum_{d:D} p$, denoting the alternatives of process p that depend on some arbitrary datum d selected from the (possibly infinite) data domain D . Using the $\sum_{d:D}$ -operator, we can interpret untimed actions $a = \sum_{t:\mathbb{R}_{\geq 0}} a \dot{c} t$, and the untimed inaction operator $\delta = \sum_{t:\mathbb{R}_{\geq 0}} \delta \dot{c} t$.

Example 1. Consider the clock process $X(t)$ that counts minutes using a clock that has an inaccuracy of at most ε , where ε is less than a second.

$$X(t:\mathbb{R}_{\geq 0}) = \sum_{d:\mathbb{R}_{\geq 0}} [d \in (1 - \varepsilon, 1 + \varepsilon)] :: \rightarrow \text{minute}(t+d) \cdot X(t+d)$$

For the (complete) axiomatisation of the operators, several auxiliary operators are required. Note that most of these auxiliary operators are never used in specifications. We here only mention the *initialisation* operator (written as $t \gg p$) denoting the behaviours of process p that can start at time t or later, where t is some point in absolute time.

Finally, we consider two processes equivalent if their behaviour is matched via *strong timed bisimulation*, see Appendix A. We write $p \dot{\leftrightarrow} t q$ if there exists a strong timed bisimulation between the two μCRL_t expressions p and q , and we say p and q are strongly timed bisimilar. We occasionally write $\mathcal{R}:p \dot{\leftrightarrow} t q$ if we have a specific strong timed bisimulation relation $\mathcal{R}$ in mind.

3 Hybrid Automata

Several variations on the definition of hybrid automata exist; here, we use the model, defined by Henzinger *et al.* [1,9], as this seems to be by far the most popular model. In Section 3.1, we introduce additional terminology; the syntax of a hybrid automaton and its semantics are defined in Section 3.2 and in Section 3.3, respectively. The embedding of hybrid automata into μCRL_t is discussed in Section 3.4.

3.1 Preliminaries

We assume a countable set $\mathcal{V}$ of *continuous variables*. These are used to model continuous behaviours of a hybrid system. To each continuous variable $x \in \mathcal{V}$, we associate a first derivative $\dot{x} \in \dot{\mathcal{V}}$ and an update variable $x' \in \mathcal{V}'$.

Continuous variables are interpreted on the real-line by *snapshot valuations*, i.e. (partial) mappings $\vartheta: \mathcal{V} \cup \dot{\mathcal{V}} \rightarrow \mathbb{R}$, and *update snapshot valuations*, i.e. (partial) mappings $\vartheta': \mathcal{V}' \rightarrow \mathbb{R}$. We denote the set of all snapshot valuations by $\mathbb{V}$ and the set of all update snapshot valuations by $\mathbb{V}'$. A *valuation* of continuous variables is a (partial) mapping f of elements of the time-line $\mathbb{R}_{\geq 0}$ to the set of snapshot valuations $\mathbb{V}$, i.e. $f: \mathbb{R}_{\geq 0} \rightarrow \mathbb{V}$. The set of all valuations is denoted by $\mathbb{F}$. We write $\mathbb{V}_V$, $\mathbb{V}'_V$ and $\mathbb{F}_V$ for (update) snapshot valuations and valuations for a finite set V of continuous variables.

We distinguish between *variable constraints*, *flow constraints* and *reset constraints*. A variable constraint is a relation between continuous variables and real numbers. The set of all variable constraints over $\mathcal{V}$ is denoted by $\Phi(\mathcal{V})$. We write $\vartheta \models \varphi$ iff $\vartheta(\varphi)$ evaluates to true, where $\vartheta(\varphi)$ is the constraint φ in which all free variables have been assigned a value according to snapshot valuation ϑ . For a valuation f , we write $f \models \varphi$ iff for all $\varepsilon: \text{dom}(f)$, $f(\varepsilon)(\varphi)$ evaluates to true. A *flow constraint* is a relation between continuous variables, their first derivatives and real numbers. The set of all flow constraints over $\mathcal{V}$ is denoted $\Theta(\mathcal{V})$. For a flow constraint $\theta \in \Theta(\mathcal{V})$ and a valuation f , we write $f \models \theta$ iff for all $\varepsilon: \text{dom}(f)$, $f(\varepsilon)(\theta)$ evaluates to true. Finally, a *reset constraint* is a relation on continuous variables, their updated counterparts and real numbers. The set of all reset constraints over $\mathcal{V}$ is denoted $R(\mathcal{V})$. For a reset constraint $\rho \in R(\mathcal{V})$, a snapshot valuation ϑ for the continuous variables in $\mathcal{V}$ and an update snapshot valuation ϑ' for their updated counterparts, we write $\vartheta' \models \rho[\vartheta]$ iff $\vartheta'(\rho)$ evaluates to true. For a finite set of continuous variables $V \subseteq \mathcal{V}$, we write $\Phi(V)$, $\Theta(V)$ and $R(V)$, respectively. We silently assume that the valuation for a variable that is not mentioned in a constraint remains unchanged.

Example 2. Typical examples of constraints are $x \leq y$ (variable constraint), $\dot{x} = y \wedge \dot{y} = 10$ (flow constraint) and $0 \leq x' \leq 5$ (reset constraint).

3.2 The Syntax of a Hybrid Automaton

We next present the formal definition of a hybrid automaton.

Definition 1. A hybrid automaton X is a tuple $\langle M, M^0, \Sigma, V, \iota, \theta, \mathcal{I}, E \rangle$, where

- M is a finite set of control modes, with initial control modes $\emptyset \subset M^0 \subseteq M$,
- Σ is a finite set of labels, and $V \subseteq \mathcal{V}$ is a finite set of continuous variables,
- $\iota: M \rightarrow \Phi(V)$ is a mapping that labels each control mode m in M with some variable constraint in $\Phi(V)$, called invariant,
- $\theta: M \rightarrow \Theta(V)$ is the flow condition function,
- $\mathcal{I}: M^0 \rightarrow \Phi(V)$ is a mapping that assigns to each initial control mode an initialisation constraint in $\Phi(V)$,
- $E \subseteq M \times \Sigma \times \Phi(V) \times R(V) \times M$ is a set of switches. We write $m \xrightarrow{\sigma, \varphi, \rho} m'$ rather than $(m, \sigma, \varphi, \rho, m') \in E$, where we say that σ is an action, φ is a guard and ρ is the reset.

We use the convention to write M, M^0 , etc. for an automaton X , and M_i, M_i^0 , etc. for automata X_i .

Remark 1. We require that for all initial modes m and all snapshot valuations ϑ , if $\vartheta(\mathcal{I}(m))$ then also $\vartheta(\iota(m))$. Automata that adhere to this requirement are called *sensible*. In the remainder of this paper, we only consider sensible automata.

A graphical notation for a hybrid automaton consists of a directed graph. The graph is annotated with additional information. The nodes (representing the control modes of the automaton) carry the information contained in the invariant and the flow condition functions. The edges (representing the switches) are annotated with actions, guards and resets.

We introduce the parallel composition operator $\parallel_s$ for hybrid automata to be able to compositionally model more complex hybrid systems from smaller components.

Definition 2. Consider the automata X_1 and X_2 , for which $V_1 \cap V_2 = \emptyset$. A synchronising hybrid automaton is the hybrid automaton $X_1 \parallel_s X_2$, defined by the tuple $\langle M_1 \times M_2, M_1^0 \times M_2^0, \Sigma_1 \cup \Sigma_2, V_1 \cup V_2, \iota, \theta, \mathcal{I}, E \rangle$, where the variable constraints ι and $\mathcal{I}$ are defined as $\iota(m, n) = \iota_1(m) \wedge \iota_2(n)$ and $\mathcal{I}(m, n) = \mathcal{I}_1(m) \wedge \mathcal{I}_2(n)$, and the flow constraint θ is defined as $\theta(m, n) = \theta_1(m) \wedge \theta_2(n)$. The set of switches E is the least set defined according to the rules of Table 1.

Note that the labels that belong to the alphabet of both hybrid automata are used for synchronisation in a parallel composition. This synchronisation is blocking, i.e. for a shared label σ , one of the hybrid automata can only execute a σ -labelled switch if the other hybrid automaton can also do so. The model we discussed here, defined by Henzinger *et al.* [1,9], does not allow for communications via shared continuous variables. Hence, the requirement $V_1 \cap V_2 = \emptyset$ in the definition of a synchronising hybrid automaton. In Section 4, we define a variant of the hybrid automaton model that does allow for communications via shared continuous variables.

Table 1. Deduction Rules for Synchronisation

$$\frac{m \xrightarrow{\sigma, \varphi, \rho} m' \quad n \xrightarrow{\sigma, \psi, \pi} n'}{(m, n) \xrightarrow{\sigma, \varphi \wedge \psi, \rho \wedge \pi} (m', n')}$$

$$\frac{m \xrightarrow{\sigma, \varphi, \rho} m' \quad \sigma \in \Sigma_1 \setminus \Sigma_2}{(m, n) \xrightarrow{\sigma, \varphi, \rho} (m', n)} \quad \frac{n \xrightarrow{\sigma, \psi, \pi} n' \quad \sigma \in \Sigma_2 \setminus \Sigma_1}{(m, n) \xrightarrow{\sigma, \psi, \pi} (m, n')}$$

3.3 The Semantics of a Hybrid Automaton

The standard interpretation, or semantics, of a hybrid automaton is defined using the underlying model of a *two-phase labelled transition system* (LTS). This means that the passing of time and the execution of an action are modelled as two separate transitions. We phrase the standard semantics using absolute time rather than relative time. As far as the semantics is concerned, this has no implications (see e.g. [4,2]) as one can think of absolute time as a clock that is never reset. The main reason for using absolute time is because this facilitates the comparison between hybrid automata and their μCRL_t -interpretations.

We start by introducing some additional notation. We write $f \upharpoonright R$ for the restriction of function f to the domain of f coinciding with R , i.e. $\text{dom}(f \upharpoonright R) = \text{dom}(f) \cap R$ and for all $\varepsilon \in \text{dom}(f \upharpoonright R)$, we have $(f \upharpoonright R)(\varepsilon) = f(\varepsilon)$. For a given valuation $f \in \mathbb{F}$, we write $\text{diff}(f)$ if f is strictly differentiable on its domain and we write $\text{cont}(f)$ if f is strictly continuous on its domain.

Definition 3. Let X be a hybrid automaton. The semantics of X is given by the LTS $X^s = \langle M \times \mathbb{V}_V \times \mathbb{R}_{\geq 0}, \mathcal{I}_{M^0}, \Sigma, \rightarrow, \vdash \rangle$, where we use the notation $\mathcal{I}_{M^0}$ to denote the set $\{(m, \vartheta, 0) \mid m \in M^0, \vartheta \in \mathbb{V}_V, \vartheta \models \mathcal{I}(m)\}$. The action transition $\rightarrow$ and the delay transition $\vdash$ are defined as the least relations satisfying the rules of Table 2.

Table 2. Operational Semantics of a Hybrid Automaton

$$\frac{m \xrightarrow{\sigma, \varphi, \rho} m' \quad \vartheta \models \iota(m) \quad \vartheta \models \varphi \quad \vartheta' \models \rho[\vartheta] \quad \vartheta' \models \iota(m')}{(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)}$$

$$\frac{f: [0, d] \rightarrow \mathbb{V}_V \quad f(0) = \vartheta \quad f(d) = \vartheta' \quad \text{cont}(f) \quad \text{diff}(f \upharpoonright (0, d)) \quad f \models \iota(m) \quad f \upharpoonright (0, d) \models \theta(m) \quad 0 \leq d}{(m, \vartheta, t) \xrightarrow{t+d} (m, \vartheta', t+d)}$$

Remark 2. Henceforth, we refer to the continuous and differentiable functions that appear in the hypothesis of the second deduction rule as *witnesses*. A state s is called *reachable* if there is a sequence of $\rightarrow$ and $\mapsto$ from an initial state to s .

Two hybrid automata are equivalent whenever their behaviour is matched via *timed bisimulation*.

Definition 4. Let X_1 and X_2 be two hybrid automata. A timed bisimulation is a relation $\mathcal{R} \subseteq (M_1 \times \mathbb{V}_{V_1} \times \mathbb{R}_{\geq 0}) \times (M_2 \times \mathbb{V}_{V_2} \times \mathbb{R}_{\geq 0})$ such that for all states (m, ϑ, t) and (n, χ, t) , the following properties are satisfied.

1. For each initial state $(m, \vartheta, 0) \in \mathcal{I}_{M_1^0}$, there is a state $(n, \chi, 0) \in \mathcal{I}_{M_2^0}$, such that $(m, \vartheta, 0) \mathcal{R} (n, \chi, 0)$, and conversely for each initial state of X_2 a related state of X_1 can be identified.
2. if $(m, \vartheta, t) \mathcal{R} (n, \chi, t)$ and $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ for some action $\sigma \in \Sigma_1$, then there is a state (n', χ', t) , such that $(n, \chi, t) \xrightarrow{\sigma} (n', \chi', t)$ and, additionally, $(m', \vartheta', t) \mathcal{R} (n', \chi', t)$ holds. Similarly for each action transition of automaton X_2 .
3. if $(m, \vartheta, t) \mathcal{R} (n, \chi, t)$ and $(m, \vartheta, t) \xrightarrow{t+d} (m, \vartheta', t+d)$ for some $d \in \mathbb{R}_{\geq 0}$, then there is a state $(n, \chi', t+d)$, such that $(n, \chi, t) \xrightarrow{t+d} (n, \chi', t+d)$ and $(m, \vartheta', t+d) \mathcal{R} (n, \chi', t+d)$. Similarly for each delay transition of automaton X_2 .

If there exists a relation $\mathcal{R}$, relating the states of X_1^s and X_2^s according to the above-listed criteria, we say the two hybrid automata X_1 and X_2 are *timed bisimilar*, denoted by $X_1 \triangleq_t X_2$. We write $\mathcal{R}: X_1 \triangleq_t X_2$ when we have a particular timed bisimulation relation $\mathcal{R}$ in mind.

3.4 Embedding Hybrid Automata in μCRL_t

For a given hybrid automaton X , we denote the set of switches, originating from a control mode $m \in M$ by E_m . For each switch $e \in E_m$, we write φ_e for the guard of switch e , σ_e for the action of switch e and ρ_e for the reset constraint of switch e . The target control mode for switch e is denoted by m_e .

Definition 5. Let X be a hybrid automaton. Let $\Theta \notin \Sigma$ be a fresh action. We write Σ_Θ to denote the set $\Sigma \cup \{\Theta\}$. The sort $\mathcal{D}_D$ represents the set of all witnesses (i.e. continuous and differentiable functions) with domain $[0, D]$ and range $\mathbb{V}_V$. The μCRL_t -interpretation $\llbracket X \rrbracket: \mathbb{P}$ is defined as $\llbracket X \rrbracket = \sum_{m \in M^0} \sum_{\vartheta: \mathbb{V}_V} [\vartheta \models \mathcal{I}(m)]:: \rightarrow X_m(0, \vartheta)$, where the control mode interpretation X_m is defined as:

$$\begin{aligned}
 X_m = & \lambda t: \mathbb{R}_{\geq 0}. \lambda \vartheta: \mathbb{V}_V. t \gg (\\
 & \sum_{D: \mathbb{R}_{\geq 0}} \sum_{F: \mathcal{D}_D} [F \models \iota(m) \wedge F(0) = \vartheta \wedge F](0, D) \models \theta(m)]:: \rightarrow \\
 & \quad \Theta \epsilon (\bar{t} + D) \cdot X_m(t + D, F(D)) \\
 & + \sum_{e \in E_m} \sum_{\zeta: \mathbb{V}_V} [\vartheta \models \varphi_e \wedge \vartheta \models \iota(m) \wedge \zeta \models \rho_e[\vartheta] \wedge \zeta \models \iota(m_e)]:: \rightarrow \\
 & \quad \sigma_e \epsilon \bar{t} \cdot X_{m_e}(t, \zeta))
 \end{aligned}$$

The embedding of a hybrid automaton is achieved by interpreting each control mode as a μCRL_t process. The continuous behaviours in a control mode are modelled using a reserved action Θ , modelling *flow transitions*. A flow transition Θ leaves a control mode m unchanged but changes the continuous variables according to the flow relation θ (i.e. after “executing” a flow transition Θ for a duration of D time units, the behaviour is again represented by a process X_m with updated time parameter and snapshot valuation). Discrete transitions σ_e on the other hand, cause a change of control mode *and*, possibly, a discontinuity in the progression of the continuous variables. Thus, the translation reflects an almost straightforward translation of the semantics of a hybrid automaton.

Example 3. To illustrate the translation, we consider the thermostat example of [10,8], see Fig. 1.

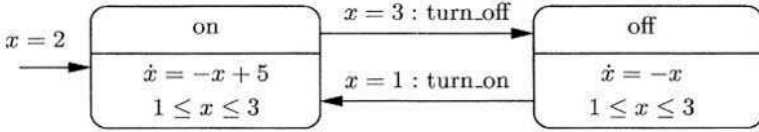


Fig. 1. Thermostat Model

The μCRL_t -interpretation of the thermostat automaton X is given below. It consists of two control mode interpretations, viz. X_{on} and X_{off} , and an initialisation, represented by process $\llbracket X \rrbracket$. The witness functions in both control mode interpretations are represented by data variable f . Since there is but one continuous variable involved, we assume the sort $\mathcal{D}_d$ contains all continuous, differentiable functions of the form $f:[0,d] \rightarrow \mathbb{R}$, rather than $f:[0,d] \rightarrow (\{x\} \rightarrow \mathbb{R})$.

$$\begin{aligned}
 \llbracket X \rrbracket &= X_{on}(0, 2) \\
 X_{on}(t:\mathbb{R}_{\geq 0}, x_c:\mathbb{R}) &= t \gg ([x_c = 3]::\rightarrow \text{turn\_off} \cdot t \cdot X_{off}(t, x_c) \\
 &\quad + \sum_{d:\mathbb{R}_{\geq 0}} \sum_{f:\mathcal{D}_d} [(\forall \varepsilon \in [0,d] 1 \leq f(\varepsilon) \leq 3) \wedge f(0) = x_c \wedge \\
 &\quad (\forall \varepsilon \in (0,d) \dot{f}(\varepsilon) = -f(\varepsilon) + 5)]::\rightarrow \Theta^c(t+d) \cdot X_{on}(t+d, f(d))) \\
 X_{off}(t:\mathbb{R}_{\geq 0}, x_c:\mathbb{R}) &= t \gg ([x_c = 1]::\rightarrow \text{turn\_on} \cdot t \cdot X_{on}(t, x_c) \\
 &\quad + \sum_{d:\mathbb{R}_{\geq 0}} \sum_{f:\mathcal{D}_d} [(\forall \varepsilon \in [0,d] 1 \leq f(\varepsilon) \leq 3) \wedge f(0) = x_c \wedge \\
 &\quad (\forall \varepsilon \in (0,d) \dot{f}(\varepsilon) = -f(\varepsilon))]]::\rightarrow \Theta^c(t+d) \cdot X_{off}(t+d, f(d)))
 \end{aligned}$$

Although the above equations seem complex, some elementary calculations can dramatically reduce the complexity of the guards for the Θ actions, as the witnesses f can actually be solved using standard analysis techniques. We leave it up to the reader to perform this simplification.

Remark 3. There is in general no translation from μCRL_t expressions to hybrid automata. First, unlike a hybrid automaton, μCRL_t expressions can terminate successfully, i.e. the expression $a^c1 \cdot b^c2$ is a valid μCRL_t expression that does

not time-lock, and for which no corresponding hybrid automaton exists. Second, most (infinite) data types (such as unbounded stacks and queues) in μCRL_t cannot be represented in a hybrid automaton.

Theorems 1 and 2, stated below, prove that the embeddings are indeed compositional and equivalence preserving. Together, they prove that μCRL_t is indeed at least as expressive as hybrid automata. In fact, hybrid automata are of strictly less expressive power than μCRL_t . Remark 3 suggests that hybrid automata correspond to the class of μCRL_t expressions that neither terminate successfully, nor encapsulate some form of infinity (in their data structures or elsewhere). Before we address Theorem 1, we first focus on the following lemma that is of use in proving Theorem 1.

Lemma 1. *Let X be a hybrid automaton, and let (m, ϑ, t) be a reachable state of X .*

1. $(m, \vartheta, t) \xrightarrow{t+d} (m, \vartheta', t+d)$ iff $X_m(t, \vartheta) \xrightarrow{\Theta}_{t+d} X_m(t+d, \vartheta')$,
2. $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ iff $X_m(t, \vartheta) \xrightarrow{\sigma}_t X_{m'}(t, \vartheta')$,
3. $\mathcal{U}_u(X_m(t, \vartheta))$ for $u \leq t$ holds by virtue of the semantics of μCRL_t ,
4. $\mathcal{U}_{t+d}(X_m(t, \vartheta))$ iff there is a ϑ' , such that $(m, \vartheta, t) \xrightarrow{t+d} (m, \vartheta', t+d)$.

Proof. Follows immediately from the semantics of a hybrid automaton and the semantics of μCRL_t (see also the proofs for Lemma 3 in Section 4).

Theorem 1. *Let X_1 and X_2 be arbitrary (sensible) hybrid automata. Then we have $X_1 \xleftrightarrow{t} X_2$ iff $\llbracket X_1 \rrbracket \xleftrightarrow{t} \llbracket X_2 \rrbracket$.*

Proof. We show the existence of a suitable (strongly) timed bisimulation relation.

“ $\Rightarrow$ ” Let $\overline{\mathcal{R}}$ be the largest timed bisimulation relation between X_1 and X_2 , i.e. $\overline{\mathcal{R}}: X_1 \xleftrightarrow{t} X_2$ relating *only* reachable states of X_1 and X_2 . Denote the reflexive, symmetric closure of $\overline{\mathcal{R}}$ by $\mathcal{R}$. Let $\mathcal{R}'$ be the relation, defined as $X_{1m}(t, \vartheta) \mathcal{R}' X_{2n}(t, \chi)$ iff $(m, \vartheta, t) \mathcal{R} (n, \chi, t)$. Then, using Lemma 1, it is easily checked that $\mathcal{R}'$ is a strongly timed bisimulation relation.

“ $\Leftarrow$ ” Let $\mathcal{R}$ be the largest strongly timed bisimulation relation relating only reachable states of the processes $\llbracket X_1 \rrbracket$ and $\llbracket X_2 \rrbracket$. Let $\mathcal{R}'$ be the relation, defined as $(m, \vartheta, t) \mathcal{R} (n, \chi, t)$ iff $X_{1m}(t, \vartheta) \mathcal{R}' X_{2n}(t, \chi)$. Then, using Lemma 1, it is easy to check that $\mathcal{R}'$ is a timed bisimulation relation.

Lemma 2. *Let X_1 and X_2 be hybrid automata, and let $((m, n), \vartheta \cup \chi, t)$ be a reachable state of the synchronising hybrid automaton $X_1 \parallel_s X_2$. For an arbitrary label set Σ , we write $\overline{\Sigma}$ for $\{\overline{\sigma} \mid \sigma \in \Sigma\}$ and we define the relabelling function $R: (\overline{\Sigma}_{1\Theta} \cap \overline{\Sigma}_{2\Theta}) \rightarrow (\Sigma_{1\Theta} \cap \Sigma_{2\Theta})$ as $R(\overline{\sigma}) = \sigma$. Abbreviate the set $\Sigma_{1\Theta} \cap \Sigma_{2\Theta}$ with H , and define the communications function γ as $\gamma(\sigma, \sigma) = \overline{\sigma}$ for all $\sigma \in H$. Then*

1. $((m, n), \vartheta \cup \chi, t) \xrightarrow{t+d} ((m, n), \vartheta' \cup \chi', t+d)$ iff $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)) \xrightarrow{\Theta}_{t+d} \rho_R \circ \partial_H(X_{1m}(t+d, \vartheta') \parallel X_{2n}(t+d, \chi'))$,

2. if $\sigma \in \Sigma_1 \cap \Sigma_2$, then $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n'), \vartheta' \cup \chi', t)$ iff $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)) \xrightarrow{\sigma}_t \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \| X_{2n'}(t, \chi'))$,
3. if $\sigma \in \Sigma_1 \setminus \Sigma_2$, then $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n), \vartheta' \cup \chi, t)$ iff $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)) \xrightarrow{\sigma}_t \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \| X_{2n}(t, \chi))$,
4. $\mathcal{U}_u(\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)))$ for $u \leq t$ holds by virtue of the semantics of μCRL_t ,
5. $\mathcal{U}_{t+d}(\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)))$ iff there are valuations ϑ', χ' , such that $((m, n), \vartheta \cup \chi, t) \xrightarrow{t+d} ((m, n), \vartheta' \cup \chi', t+d)$.

Proof. Follows from the semantics of the parallel operators in μCRL_t and for hybrid automata (see also the proofs for Lemma 4).

Theorem 2. Let X_1 and X_2 be hybrid automata, let R be the relabelling function and H the encapsulation set, both defined in Lemma 2. Then, we have $\llbracket X_1 \rrbracket_s X_2 \Downarrow \Leftrightarrow_t \rho_R \circ \partial_H(\llbracket X_1 \rrbracket \| \llbracket X_2 \rrbracket)$.

Proof. The equivalence closure of the relation $\mathcal{R}$, defined as

$$\mathcal{R} = \{ \langle X_{1,2(m,n)}(t, \vartheta \cup \chi), \rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)) \rangle \mid \text{for all reachable states } ((m, n), \vartheta \cup \chi, t) \text{ of } X_1 \|_s X_2 \}$$

is easily shown to be a strongly timed bisimulation relation between $\llbracket X_1 \rrbracket_s X_2$ and $\rho_R \circ \partial_H(\llbracket X_1 \rrbracket \| \llbracket X_2 \rrbracket)$, using lemmata 1 and 2.

Remark 4. The time steps in the timed transition system, induced by a hybrid automaton, in general do not adhere to properties such as *time-additivity* (also known as *time-continuity*) and *time-determinism*, placing them outside the scope of classical timed systems. In our translation, continuous flows are hence treated as true behaviours of a system. In general, we cannot abstract from these continuous behaviours, but for special classes of hybrid automata, the induced timed transition system adheres to the aforementioned properties. In such cases, the translation of hybrid automata into μCRL_t defined in [12], is more appropriate, as it circumvents the use of the dedicated Θ action. Note that also the thermostat example, discussed in this section, would fall in the class of systems that admit a more direct translation, as its continuous behaviour is fully deterministic.

4 Enhanced Hybrid Automata

The standard model of a hybrid automaton does not support communications via continuous variables. As communications via shared continuous variables are often used in disciplines, other than computer science, the addition of such capabilities is easily motivated. There are many ways in which these features can be added to the standard model of hybrid automata. We choose to study a minor extension of the standard model that offers these capabilities with only a subtle change in the syntax and the semantics.

4.1 Semantics of an Enhanced Hybrid Automaton

We extend hybrid automata as follows. An *enhanced hybrid automaton* is a hybrid automaton with two sets of continuous variables, i.e. P is the set of public continuous variables and I is the set of private continuous variables. We require $P \cap I = \emptyset$, and $V = P \cup I$. The semantics of an enhanced hybrid automaton is defined below. First, we introduce a shorthand. For a function $f:A \rightarrow (B \rightarrow C)$, we write $f[R]$ for the function $f:A \rightarrow (B \cap R \rightarrow C)$, which is defined as $f(a)(b) = (f[R])(a)(b)$ for all $a \in A$ and $b \in B \cap R$. As a convention, we define $f[\emptyset = \text{dom}(f)]$.

Definition 6. Let X be an enhanced hybrid automaton. The semantics of X is given by the LTS $X^s = \langle M \times \mathbb{V}_V \times \mathbb{R}_{\geq 0}, \mathcal{I}_{M^0}, \Sigma, \rightarrow, \mapsto \rangle$. The action transition $\rightarrow$ and the flow transition $\mapsto$ are defined as the least relations satisfying the rules of Table 3. The state space of X^s consists of the set of states that can be reached from the initial states.

Table 3. Operational Semantics of an Enhanced Hybrid Automaton

$m \xrightarrow{\sigma, \varphi, \rho} m'$	$\vartheta \models \iota(m)$	$\vartheta \models \varphi$	$\vartheta' \models \rho[\vartheta]$	$\vartheta' \models \iota(m')$	
$(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$					
$f:[0, d] \rightarrow \mathbb{V}_V$	$f(0) = \vartheta$	$f(d) = \vartheta'$	$\text{cont}(f)$		
$\text{diff}(f)[0, d)$	$f \models \iota(m)$	$f[0, d) \models \theta(m)$	$0 \leq d$		
$(m, \vartheta, t) \xrightarrow{f[P]} (m, \vartheta', t + d)$					

Rather than mere time steps, an enhanced hybrid automaton can execute *flow* transitions. These consist of a witness for the public continuous variables. We say two enhanced hybrid automata are equivalent if there exists a *flow bisimulation* relation, relating both systems (see below). Basically, a flow bisimulation relates all behaviours, i.e. both the (public) continuous behaviours and the discrete behaviours.

Definition 7. Let X_1 and X_2 be two enhanced hybrid automata, for which $P = P_1 = P_2$. A relation $\mathcal{R} \subseteq (M_1 \times \mathbb{V}_{V_1} \times \mathbb{R}_{\geq 0}) \times (M_2 \times \mathbb{V}_{V_2} \times \mathbb{R}_{\geq 0})$ is a flow bisimulation iff for all states (m, ϑ, t) and (n, χ, t) , the following properties are satisfied.

1. For each initial state $(m, \vartheta, 0) \in \mathcal{I}_{M_1^0}$, there is a state $(n, \chi, 0) \in \mathcal{I}_{M_2^0}$, such that $(m, \vartheta, 0) \mathcal{R} (n, \chi, 0)$, and conversely for each initial state of X_2 a related state of X_1 can be identified.

2. if $(m, \vartheta, t) \mathcal{R}(n, \chi, t)$ and $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ for some action $\sigma \in \Sigma_1$, then there is a state (n', χ', t) , such that $(n, \chi, t) \xrightarrow{\sigma} (n', \chi', t)$ and, additionally, $(m', \vartheta', t) \mathcal{R}(n', \chi', t)$ holds. Similarly for each action transition of automaton X_2 .
3. if $(m, \vartheta, t) \mathcal{R}(n, \chi, t)$ and $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', t + d)$ for some $d \in \mathbb{R}_{\geq 0}$, then there is a state $(n, \chi', t + d)$, such that $(n, \chi, t) \xrightarrow{f} (n, \chi', t + d)$ and $(m, \vartheta', t + d) \mathcal{R}(n, \chi', t + d)$. Similarly for each flow transition of automaton X_2 .

If there exists a relation $\mathcal{R}$, relating the states of X_1^s and X_2^s according to the above-listed criteria, we say the two enhanced hybrid automata X_1 and X_2 are flow bisimilar, denoted by $X_1 \Leftrightarrow_f X_2$. We write $\mathcal{R}: X_1 \Leftrightarrow_f X_2$ when we have a particular flow bisimulation $\mathcal{R}$ in mind.

Notice that the duration of a flow transition is deduced from the length of the domain of the witness, which is a closed interval.

Example 4. Consider two enhanced hybrid automata X_1 and X_2 (see Figure 2) with $P_1 = P_2 = \{x\}$. If we interpret X_1 and X_2 as ordinary hybrid automata, we can deduce that $X_1 \Leftrightarrow_t X_2$. This is confirmed by the timed bisimulation relation $\mathcal{R} = \{((X, d, t + d), (Y, d^2, t + d)) \mid 0 \leq d \leq 1 \wedge t \in \mathbb{N}\}$. However, we do not have $X_1 \Leftrightarrow_f X_2$, as from state $(X, 0, 0)$, we can execute a flow transition $F: [0, 1] \rightarrow (\{x\} \rightarrow \mathbb{R})$, defined as $F(\varepsilon)(x) = \varepsilon$. From state $(Y, 0, 0)$ we cannot mimic this transition.

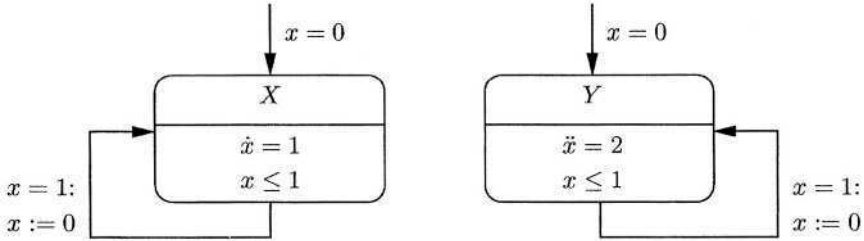


Fig. 2. Two simple Hybrid Systems

Proposition 1. For all enhanced hybrid automata X_1 and X_2 , we have

1. whenever $X_1 \Leftrightarrow_f X_2$ then also $X_1 \Leftrightarrow_t X_2$,
2. whenever $P_1 = P_2 = \emptyset$, we have $X_1 \Leftrightarrow_f X_2$ iff $X_1 \Leftrightarrow_t X_2$,
3. by definition, whenever $P_1 \neq P_2$, we have $X_1 \not\Leftrightarrow_f X_2$.

The composition of two enhanced hybrid automata is only well-defined when the two automata *match*. We say two enhanced hybrid automata X_1 and X_2 match iff $P_1 = P_2$, $I_1 \cap I_2 = \emptyset$, and for all switches $e_i \in E_1 \cup E_2$ we require that whenever ρ_{e_i} contains variables from $P_1 (= P_2)$ then $\sigma_{e_i} \in \Sigma_1 \cap \Sigma_2$. In

other words, two enhanced hybrid automata match iff their private variables are disjoint, their public variables are the same and all public variables are reset only via the execution of a shared discrete action. The definition of a *synchronising enhanced hybrid automaton* $X_1 \parallel_s X_2$ is the same as the definition of a synchronising hybrid automaton, where $P_{1,2} = P_1 = P_2$ and $I_{1,2} = I_1 \cup I_2$.

4.2 Embedding Enhanced Hybrid Automata in μCRL_t

We subsequently show that we can embed enhanced hybrid automata in μCRL_t .

Definition 8. Let X be an enhanced hybrid automaton. Let $\Theta \notin \Sigma$ be a reserved action. We write Σ_Θ to denote the set $\Sigma \cup \{\Theta\}$. The μCRL_t -interpretation $\llbracket X \rrbracket : \mathbb{P}$ is defined as $\llbracket X \rrbracket = \sum_{m \in M^0} \sum_{\vartheta : \mathbb{V}_V} [\vartheta \models \mathcal{I}(m)] :: \rightarrow X_m(0, \vartheta)$, where the control mode interpretation X_m is defined as:

$$\begin{aligned} X_m = & \lambda t : \mathbb{R}_{\geq 0}. \lambda \vartheta : \mathbb{V}_V. t \gg \\ & (\sum_{D : \mathbb{R}_{> 0}} \sum_{F : \mathcal{D}_D} [F \models \iota(m) \wedge F(0) = \vartheta \wedge F](0, D) \models \theta(m)] :: \rightarrow \\ & \Theta(F[P] \cdot t \cdot ((t+D) \gg X_m(t+D, F(D))) \\ & + \sum_{e \in E_m} \sum_{\zeta : \mathbb{V}_V}. [\vartheta \models \varphi_e \wedge \vartheta \models \iota(m) \wedge \zeta \models \rho_e[\vartheta] \wedge \zeta \models \iota(m_e)] :: \rightarrow \\ & \sigma_e(\zeta[P] \cdot t \cdot X_{m_e}(t, \zeta)) \end{aligned}$$

The embedding is similar to the embedding of standard hybrid automata into μCRL_t . The actions σ_e are parameterised, carrying information about the new snapshot valuation for the public continuous variables. This ensures that components, possibly participating in a parallel composition, have the same view on the public continuous variables. Unlike the flow transitions in Def. 5, the (parameterised) flow transitions $\Theta(F[P])$ are executed at time t , marking the *start* of a flow transition. The end of the flow transition is marked by the time-reinitialisation to time $t + D$. Note that this initialisation is in fact redundant, as it is already part of the process $X_m(t + D, F(D))$.

Lemma 3. Let X be an enhanced hybrid automaton, and let (m, ϑ, t) be a reachable state of X .

1. $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', t+d)$ iff $X_m(t, \vartheta) \xrightarrow{\Theta(f)}_t (t+d) \gg X_m(t+d, \vartheta')$,
2. $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ iff $X_m(t, \vartheta) \xrightarrow{\sigma(\vartheta'[P])}_t X_{m'}(t, \vartheta')$,
3. $\mathcal{U}_u(X_m(t, \vartheta))$ for $u \leq t$ holds by virtue of the semantics of μCRL_t ,
4. $\mathcal{U}_{t+d}(X_m(t, \vartheta))$ iff there are both a valuation ϑ' and a witness f such that $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', t+d)$.

Proof. We sketch the proofs for the first two statements of Lemma 3. Let X be an enhanced hybrid automaton, and let (m, ϑ, t) be a reachable state of X .

1. Assume that $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', t+d)$ for some state $(m, \vartheta', t+d)$. According to the rules in Table 3, we then have the existence of a witness $f_b : [0, d] \rightarrow \mathbb{V}_V$, with $f_b[P] = f$, $f_b(0) = \vartheta$, $f_b(d) = \vartheta'$, $f_b \models \iota(m)$ and $f_b[0, d] \models \theta(m)$. Thus, there is a valuation ν for the variable D of sort $\mathbb{R}_{\geq 0}$ and F of sort $\mathcal{D}_D$ in

the interpretation given in Def. 8 that is such that $F \models \iota(m)$, $F(0) = \vartheta$, $F[0, D] \models \theta(m)$ is satisfied and for which the interpretation of $F[P]$ is exactly f . It then follows that we have $X_m(t, \vartheta) \xrightarrow{\Theta(f)}_t (t+d) \gg X_m(t+d, \vartheta')$. The converse follows the same line of reasoning in reverse order.

2. Assume that $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ for some state (m', ϑ', t) . Using the rules of Table 3, we then have the existence of a switch $m \xrightarrow{\sigma, \varphi, \rho} m'$, such that $\vartheta \models \iota(m)$, $\vartheta \models \varphi$, $\vartheta' \models \rho[\vartheta]$ and $\vartheta' \models \iota(m')$. This means that we can find a valuation ν for the variables e and ζ in the interpretation given in Def. 8, that is such that $\vartheta \models \varphi_e$, $\vartheta \models \iota(m)$, $\zeta \models \rho_e[\vartheta]$ and $\zeta \models \iota(m_e)$ is satisfied. Then we also have $X_m(t, \vartheta) \xrightarrow{\sigma(\zeta[P])}_t X_{m'}(t, \vartheta')$. The converse follows the same line of reasoning in reverse order.

Theorem 3. *Let X_1 and X_2 be arbitrary (sensible) enhanced hybrid automata. Then we have $\llbracket X_1 \rrbracket \Leftrightarrow_t \llbracket X_2 \rrbracket$ iff $X_1 \Leftrightarrow_f^t X_2$.*

Proof. We only sketch one direction of the proof. The other direction of the proof is similar. Let X_1 and X_2 be two enhanced hybrid automata with $P_1 = P_2$, and let $\overline{\mathcal{R}}$ be the largest flow bisimulation relation between X_1 and X_2 relating reachable states only. Let $\mathcal{R}$ be the reflexive, symmetric closure of $\overline{\mathcal{R}}$. Define $\mathcal{R}'$ as the smallest relation, such that $X_{1m}(t, \vartheta) \mathcal{R}' X_{2n}(t, \chi)$ iff $(m, \vartheta, t) \mathcal{R}(n, \chi, t)$. We show that $\mathcal{R}'$ is a strongly timed bisimulation relation. By definition, $\mathcal{R}'$ is symmetric. Assume $X_{1m}(t, \vartheta) \mathcal{R}' X_{2n}(t, \chi)$. Since we know that neither $\llbracket X_1 \rrbracket$, nor $\llbracket X_2 \rrbracket$ terminate successfully, it suffices to investigate the following two cases:

1. Suppose $X_{1m}(t, \vartheta) \xrightarrow{\sigma(\vartheta')^P}_t X_{1m'}(t, \vartheta')$ for some m', ϑ' and σ . By Lemma 3, we know that also $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$. Since $\mathcal{R}$ is a flow bisimulation, we also have $(n, \chi, t) \xrightarrow{\sigma} (n', \chi', t)$ for some n' and χ' , such that $(m', \vartheta', t) \mathcal{R}(n', \chi', t)$. Since we can always execute a flow transition of duration 0, reporting the values of the public continuous variables, we know that $\vartheta'[P] = \chi'[P]$. Again, by Lemma 3, we find that also $X_{2n}(t, \chi) \xrightarrow{\sigma(\chi')^P}_t X_{2n'}(t, \chi')$, and $X_{1m'}(t, \vartheta') \mathcal{R}' X_{2n'}(t, \chi')$. The case for action $\Theta(f)$ follows a similar line of reasoning.
2. Let $r \in \mathbb{R}_{\geq 0}$ be an arbitrary point of time. Then, by Lemma 3, we find for $r \leq t$ that $\mathcal{U}_r(X_{1m}(t, \vartheta))$ holds, if and only if $\mathcal{U}_r(X_{2n}(t, \chi))$. For $r \geq t$, we reason as follows. From $X_{1m}(t, \vartheta) \mathcal{R}' X_{2n}(t, \chi)$, it follows that we have $(m, \vartheta, t) \mathcal{R}(n, \chi, t)$. Now, assume we have $\mathcal{U}_r(X_{1m}(t, \vartheta))$. Then, we know that there is a valuation ϑ' and a flow f , such that $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', r)$. Since $\mathcal{R}$ is a flow bisimulation, we also have $(n, \chi, t) \xrightarrow{f} (n, \chi', r)$, for some χ' . But then by Lemma 3, we also find $\mathcal{U}_r(X_{2n}(t, \chi))$.

The communication mechanism of μCRL_t (which requires that not only the actions should be able to communicate, but also their data-parameters should agree) ensures that flow transitions in each component participating in a parallel composition all have the same witnesses, and, thus duration (i.e. the length of

the domain of the function F in the translation, which equals D). Thus, also Theorem 2 has its counterpart for enhanced hybrid automata. This is formalised in Theorem 4. The proof of this theorem relies on the following Lemma.

Lemma 4. *Let X_1 and X_2 be two matching enhanced hybrid automata and let $((m, n), \vartheta \cup \chi, t)$ be a reachable state of the synchronising enhanced hybrid automaton $X_1 \parallel_s X_2$. Let R be the relabelling function and H be the encapsulation set of Lemma 2, and write P for $P_1 (= P_2)$. Then*

1. $((m, n), \vartheta \cup \chi, t) \xrightarrow{f} ((m, n), \vartheta' \cup \chi', t+d)$ iff
 $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)) \xrightarrow{\Theta(f)}_t \rho_R \circ \partial_H(X_{1m}(t+d, \vartheta') \parallel X_{2n}(t+d, \chi'))$,
2. if $\sigma \in \Sigma_1 \cap \Sigma_2$, then $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n'), \vartheta' \cup \chi', t)$ iff
 $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)) \xrightarrow{\sigma(\vartheta']^P}_t \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \parallel X_{2n'}(t, \chi'))$,
3. if $\sigma \in \Sigma_1 \setminus \Sigma_2$, then $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n), \vartheta' \cup \chi, t)$ iff
 $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)) \xrightarrow{\sigma(\vartheta']^P}_t \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \parallel X_{2n}(t, \chi))$,
4. $\mathcal{U}_u(\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)))$ for $u \leq t$ holds by virtue of the semantics of μCRL_t ,
5. $\mathcal{U}_{t+d}(\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)))$ iff there are valuations ϑ', χ' , such that
 $((m, n), \vartheta \cup \chi, t) \xrightarrow{f} ((m, n), \vartheta' \cup \chi', t+d)$.

Proof. We only sketch the proof of the first three properties of Lemma 4. The remaining properties are similar to the ones treated here. Let X_1 and X_2 be matching enhanced hybrid automata, and let $((m, n), \vartheta \cup \chi, t)$ be a reachable state of the synchronising enhanced hybrid automaton $X_1 \parallel_s X_2$. Let H and R be as defined in Lemma 2.

1. Assume $((m, n), \vartheta \cup \chi, t) \xrightarrow{f} ((m, n), \vartheta' \cup \chi', t+d)$. This means that with valuations ϑ and χ , both X_1 and X_2 should be able to portray flow transitions f for exactly d seconds in control modes m and n respectively. This means that $(m, \vartheta, t) \xrightarrow{f} (m, \vartheta', t+d)$ and $(n, \chi, t) \xrightarrow{f} (n, \chi', t+d)$. We then know that the location interpretations X_{1m} and X_{2n} can perform a flow transition (see Lemma 3). Therefore, $X_{1m}(t, \vartheta) \xrightarrow{\Theta(f)}_t (t+d) \gg X_{1m}(t+d, \vartheta')$ and $X_{2n}(t, \chi) \xrightarrow{\Theta(f)}_t (t+d) \gg X_{2n}(t+d, \chi')$. Since we have, by definition, that $(t+d) \gg X_{1m}(t+d, \vartheta')$ is strongly timed bisimilar with $X_{1m}(t+d, \vartheta')$, and likewise for X_{2n} , we then also have $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \parallel X_{2n}(t, \chi)) \xrightarrow{\Theta(f)}_t \rho_R \circ \partial_H(X_{1m}(t+d, \vartheta') \parallel X_{2n}(t+d, \chi'))$. The converse follows the same reasoning in reverse order.
2. Let $\sigma \in \Sigma_1 \cap \Sigma_2$. Assume $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n'), \vartheta' \cup \chi', t)$. This means that with snapshot valuations ϑ and χ , both X_1 and X_2 have a switch that is enabled for control modes m and n respectively. Moreover, the reset constraints for these switches must agree on the valuations ϑ' and χ' for the variables from set P , i.e. $\vartheta']^P = \chi']^P$. Thus, we also know that $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$ and $(n, \chi, t) \xrightarrow{\sigma} (n', \chi', t)$. But then, by Lemma 3, also $X_{1m}(t, \vartheta) \xrightarrow{\sigma(\vartheta']^P}_t X_{1m'}(t, \vartheta')$ and $X_{2n}(t, \chi) \xrightarrow{\sigma(\chi']^P}_t X_{2n'}(t, \chi')$. Thus,

also $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)) \xrightarrow{\sigma(\vartheta']^P} \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \| X_{2n'}(t, \chi'))$.
The converse again follows the same reasoning in reverse order.

3. Let $\sigma \in \Sigma_1 \setminus \Sigma_2$. Assume $((m, n), \vartheta \cup \chi, t) \xrightarrow{\sigma} ((m', n'), \vartheta' \cup \chi', t)$. This means that with snapshot valuation ϑ , X_1 has a switch between control mode m and m' , that is enabled for control mode m . Moreover the reset constraint of this switch agrees with the valuation ϑ' . Since X_1 and X_2 match, we have $\vartheta' \upharpoonright P = \chi \upharpoonright P$ (this is a requirement on the reset constraint of the switch that follows from the definition of matching). Thus, we also have $(m, \vartheta, t) \xrightarrow{\sigma} (m', \vartheta', t)$.

By Lemma 3, we then also have $X_{1m}(t, \vartheta) \xrightarrow{\sigma(\vartheta']^P} X_{1m'}(t, \vartheta')$. Moreover, by Lemma 3, we also have $\mathcal{U}_t(X_{2n}(t, \chi))$. Combining both, we then find that

also $\rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \vartheta)) \xrightarrow{\sigma(\vartheta']^P} \rho_R \circ \partial_H(X_{1m'}(t, \vartheta') \| X_{2n'}(t, \chi'))$.
The converse follows the same line of reasoning in reverse order.

Theorem 4. *Let X_1 and X_2 be arbitrary, matching enhanced hybrid automata. Then we have $\langle X_1 \|_s X_2 \rangle \xleftrightarrow{\tau} \rho_R \circ \partial_H(\langle X_1 \rangle \| \langle X_2 \rangle)$, where R , H and communications function γ are as defined in Lemma 2.*

Proof. Define the relation $\mathcal{R}$ as

$$\mathcal{R} = \{ \langle X_{1,2(m,n)}(t, \vartheta \cup \chi), \rho_R \circ \partial_H(X_{1m}(t, \vartheta) \| X_{2n}(t, \chi)) \rangle \mid \text{for all reachable states } ((m, n), \vartheta \cup \chi, t) \text{ of } X_1 \|_s X_2 \}$$

Using lemmata 3 and 4, the equivalence closure of $\mathcal{R}$ is easily shown to be a strongly timed bisimulation relation between $\langle X_1 \|_s X_2 \rangle$ and $\rho_R \circ \partial_H(\langle X_1 \rangle \| \langle X_2 \rangle)$

Remark 5. The above results indicate that μCRL_t is at least as expressive as enhanced hybrid automata. The fact that the μCRL_t -examples of Remark 3 still cannot be translated to enhanced hybrid automata in fact show that μCRL_t is also strictly more expressive than enhanced hybrid automata.

5 Conclusions

In this paper, we investigated hybrid automata [1,9] and *enhanced hybrid automata*. The latter is a minor extension of standard hybrid automata, offering basic support for communication via shared continuous variables. We showed that both modelling formalisms can in fact be embedded in a real-time process algebra, called μCRL_t [5,11]. The embedding of (enhanced) hybrid automata in μCRL_t shows that discrete behaviours and continuous behaviours must be treated similarly.

Several interesting issues are still left open for investigation. First, it may be worthwhile to investigate several notions of *abstraction*. It is not clear whether the techniques for abstraction, currently investigated for timed process algebras, also apply to the Θ actions. Second, it is realistic to introduce a notion of abstraction on the parameters for the parameterised actions. For instance, in enhanced hybrid automata, it may be useful to abstract from public continuous variables. It is not clear how this can be done in μCRL_t .

Acknowledgements. The author would like to thank Frits Vaandrager (University of Nijmegen), whose questions concerning two particular hybrid automata and their possible μCRL_t interpretations led to the results described in this paper. The reviewers of IFM 2004 are thanked for their constructive comments.

References

1. R. Alur, T.A. Henzinger, and P.-H. Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, 1996.
2. J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing*. EATCS Monograph. Springer-Verlag, 2002.
3. V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. PhD thesis, Eindhoven University of Technology, March 2002.
4. F. Corradini. Absolute versus relative time in process algebras. In C. Palamidessi and J. Parrow, editors, *Proceedings of EXPRESS'97*, volume 7 of *ENTCS*, pages 113–132. Elsevier, 1997.
5. J.F. Groote. The syntax and semantics of timed μCRL . Software Engineering Report SEN-R9709, CWI, June 1997.
6. J.F. Groote and A. Ponse. The syntax and semantics of μCRL . In A. Ponse, C. Verhoef, and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer Verlag, 1995.
7. J.F. Groote and M.A. Reniers. Algebraic process verification. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 17, pages 1151–1208. Elsevier (North-Holland), 2001.
8. J.F. Groote and J.J. van Wamel. Analysis of three hybrid systems in timed μCRL . *Journal of Logic and Algebraic Programming*, 39:215–247, 2001.
9. T.A. Henzinger. The theory of hybrid automata. In *the Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS 96)*, pages 278–292, 1996.
10. T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Software Tools for Technology Transfer*, 1:110–122, 1997.
11. M.A. Reniers, J.F. Groote, M.B. van der Zwaag, and J. van Wamel. Completeness of timed μCRL . *Fundamenta Informaticae*, 50(3–4):361–402, 2002.
12. T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. PhD thesis, Eindhoven University of Technology, Februari 2003.

A Operational Semantics for μCRL_t

The operational semantics of μCRL_t is defined in two steps. First, we discuss the semantics of $p\text{CRL}_t$, which is the concurrent-free fragment of μCRL_t . This means that the available operators are from the set $\{+, \cdot, \sum_d, \triangleleft, \triangleright, \cdot, \gg\}$. Due to the presence of data in process terms, the semantics of $p\text{CRL}_t$ expressions is not defined on process terms, but on *interpreted* process terms, called *processes*. We assume a set *Act* of action declarations of the form $a:s_1 \times \dots \times s_n \rightarrow \mathbb{P}$, where s_i are sort symbols from the data signature. For each sort symbol s , we assume a data algebra with universe $\mathcal{D}_s$. The sort symbol $\mathbb{T}$ stands for an arbitrary, totally ordered time domain with least element 0 .

Definition 9. The set of $pCRL_t$ -processes $\mathcal{P}$ and the set of $pCRL_t$ -actions $\mathcal{A}$ are defined below, where the set of processes $\mathcal{P} = \bigcup_{i=0}^{\omega} \mathcal{P}^i$.

$$\begin{aligned}\mathcal{A} &= \{a(d_1, \dots, d_n) \mid a:s_1 \times \dots \times s_n \in Act, d_i \in \mathcal{D}_{s_i}\} \\ \mathcal{P}^0 &= \mathcal{A} \cup \{\delta\} \\ \mathcal{P}^{n+1} &= \mathcal{P}^n \cup \{p \cdot q, \sum P, p \dot{+} t, t \gg p \mid p, q \in \mathcal{P}^n, \emptyset \subset P \subseteq \mathcal{P}^n, t \in \mathcal{D}_T\}\end{aligned}$$

Definition 10. We assume a valuation function ν for the data and time variables occurring in process expressions. Expressions of $pCRL_t$ are interpreted according to the interpretation function we define below, yielding processes.

$$\begin{aligned}\llbracket a(d_1, \dots, d_n) \rrbracket^\nu &= a(\llbracket d_1 \rrbracket^\nu, \dots, \llbracket d_n \rrbracket^\nu) \\ \llbracket \delta \rrbracket^\nu &= \delta \\ \llbracket p + q \rrbracket^\nu &= \sum \{\llbracket p \rrbracket^\nu, \llbracket q \rrbracket^\nu\} \\ \llbracket p \cdot q \rrbracket^\nu &= \llbracket p \rrbracket^\nu \cdot \llbracket q \rrbracket^\nu \\ \llbracket \sum_v p \rrbracket^\nu &= \sum \{\llbracket p \rrbracket^{\nu'} \mid \nu(v') = \nu'(v') \text{ for all } v' \neq v\} \\ \llbracket p \triangleleft b \triangleright q \rrbracket^\nu &= \begin{cases} \llbracket p \rrbracket^\nu & \text{if } \llbracket b \rrbracket^\nu \text{ is true} \\ \llbracket q \rrbracket^\nu & \text{if } \llbracket b \rrbracket^\nu \text{ is false} \end{cases} \\ \llbracket t \gg p \rrbracket^\nu &= \llbracket t \rrbracket^\nu \gg \llbracket p \rrbracket^\nu \\ \llbracket p \dot{+} t \rrbracket^\nu &= \llbracket p \rrbracket^\nu \dot{+} \llbracket t \rrbracket^\nu\end{aligned}$$

Definition 11. To each expression in the theory of $pCRL_t$, we associate a time-stamped labelled transition system $\mathcal{L} = \langle \mathcal{P}, \mathcal{A}, \rightarrow, \sqrt{}, \mathcal{U} \rangle$, where $\mathcal{P}$ and $\mathcal{A}$ are obtained by interpreting $pCRL_t$ -expressions. The transition relation $\rightarrow$, the ultimate delay operator $\mathcal{U}$ and the termination-upon-transition relation $\sqrt{}$ are defined using Plotkin-style rules in Table 4.

The operational semantics of μCRL_t extends on the operational semantics of $pCRL_t$ by adding rules for the operators from the set $\{\parallel, \llbracket \rrbracket, |, \ll, \rho_R, \partial_H\}$. We again first interpret μCRL_t -expressions into μCRL_t -processes. The set $\mathcal{A}$ of μCRL_t -actions is taken to be the same as the set of $pCRL_t$ -actions.

Definition 12. We redefine the set of processes $\mathcal{P} = \bigcup_{i=0}^{\omega} \mathcal{P}^i$, where $\mathcal{P}^n$ is defined inductively below.

$$\begin{aligned}\mathcal{P}^0 &= \mathcal{A} \cup \{\delta\} \\ \mathcal{P}^{n+1} &= \mathcal{P}^n \cup \{p \cdot q, \sum P, p \parallel q, p \llbracket q \rrbracket, p | q, p \dot{+} t, t \gg p, p \ll q, \rho_R(p), \partial_H(p) \mid \\ &\quad p, q \in \mathcal{P}^n, \emptyset \subset P \subseteq \mathcal{P}^n, t \in \mathcal{D}_T, H \subseteq Act, R: Act \rightarrow Act\}\end{aligned}$$

Definition 13. Let ν be a valuation of the data and time variables occurring in μCRL_t -expressions. We interpret μCRL_t -expressions according to the function $\llbracket - \rrbracket^\nu$, defined for $pCRL_t$ -expressions, extended with the identities below.

$$\begin{aligned}\llbracket p \parallel q \rrbracket^\nu &= \llbracket p \rrbracket^\nu \parallel \llbracket q \rrbracket^\nu \\ \llbracket p \llbracket q \rrbracket \rrbracket^\nu &= \llbracket p \rrbracket^\nu \llbracket \llbracket q \rrbracket^\nu \rrbracket^\nu \\ \llbracket p | q \rrbracket^\nu &= \llbracket p \rrbracket^\nu | \llbracket q \rrbracket^\nu \\ \llbracket p \ll q \rrbracket^\nu &= \llbracket p \rrbracket^\nu \ll \llbracket q \rrbracket^\nu \\ \llbracket \partial_H(p) \rrbracket^\nu &= \partial_H(\llbracket p \rrbracket^\nu) \\ \llbracket \rho_R(p) \rrbracket^\nu &= \rho_R(\llbracket p \rrbracket^\nu)\end{aligned}$$

Table 4. Transition relation, the termination-upon-transition relation and the ultimate delay for $pCRL_t$ processes, where $a \in \mathcal{A}$, $p, q, p' \in \mathcal{P}$, $t, t' \in \mathcal{D}_T$ and $P \subseteq \mathcal{P}$.

$a \xrightarrow{a}_t \checkmark$	$\frac{p \xrightarrow{a}_t \checkmark}{p \cdot t \xrightarrow{a}_t \checkmark}$	$\frac{p \xrightarrow{a}_t \checkmark}{\sum(P \cup \{p\}) \xrightarrow{a}_t \checkmark}$	$\frac{p \xrightarrow{a}_t \checkmark \quad t' \leq t}{t' \gg p \xrightarrow{a}_t \checkmark}$
$\frac{p \xrightarrow{a}_t p'}{p \cdot t \xrightarrow{a}_t p'}$	$\frac{p \xrightarrow{a}_t p'}{\sum(P \cup \{p\}) \xrightarrow{a}_t p'}$	$\frac{p \xrightarrow{a}_t p'}{p \cdot q \xrightarrow{a}_t p' \cdot q}$	
$\frac{p \xrightarrow{a}_t p' \quad t' \leq t}{t' \gg p \xrightarrow{a}_t p'}$	$\frac{p \xrightarrow{a}_t \checkmark}{p \cdot q \xrightarrow{a}_t t \gg q}$		
$\mathcal{U}_t(a)$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(p \cdot q)}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(\sum(P \cup \{p\}))}$	$\frac{\mathcal{U}_t(p) \quad t \leq t'}{\mathcal{U}_t(p \cdot t')}$
	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(t' \gg p)}$	$\frac{t \leq t'}{\mathcal{U}_t(t' \gg p)}$	

Definition 14. We associate a time-stamped labelled transition system $\mathcal{L} = \langle \mathcal{P}, \mathcal{A}, \rightarrow, \checkmark, \mathcal{U} \rangle$ to each expression in the theory of μCRL_t , where $\mathcal{P}$ and $\mathcal{A}$ are obtained by interpreting μCRL_t -expressions. The transition relation $\rightarrow$ and the termination-upon-transition relation $\checkmark$ are defined in Table 5, and the ultimate delay operator $\mathcal{U}$ is defined in Table 6.

As usual, the process algebra μCRL_t uses an axiom system to equate process expressions. The equality, expressed by the axioms, is interpreted as *strong timed bisimulation* on processes.

Definition 15. A symmetric relation $\mathcal{R} \subseteq \mathcal{P} \times \mathcal{P}$ on processes is a *strong timed bisimulation* iff for all processes p and q , the following three properties are satisfied.

1. if $p \mathcal{R} q$ and $p \xrightarrow{a}_t p'$ for some $a \in \mathcal{A}$ and $t \in \mathcal{D}_T$, then there exists a process q' , such that $q \xrightarrow{a}_t q'$ and $p' \mathcal{R} q'$ holds.
2. if $p \mathcal{R} q$ and $p \xrightarrow{a}_t \checkmark$ for some $a \in \mathcal{A}$ and $t \in \mathcal{D}_T$, then also $q \xrightarrow{a}_t \checkmark$.
3. if $p \mathcal{R} q$, then $\mathcal{U}_t(p)$ iff $\mathcal{U}_t(q)$ for all $t \in \mathcal{D}_T$.

Two μCRL_t processes p and q are *strongly timed bisimilar*, denoted by $p \triangleq_t q$, iff there exists a strong timed bisimulation relation $\mathcal{R}$, such that $p \mathcal{R} q$. By abuse of notation, we write $p \triangleq_t q$ for μCRL_t expressions p and q , where we actually mean $\llbracket p \rrbracket^\nu \triangleq_t \llbracket q \rrbracket^\nu$ for all data valuations ν .

For an overview of the axiom system, several meta-results, such as (relative) completeness and soundness of the axiomatisation, and a more verbose explanation of the operators (including many of the auxiliary operators), we refer to [11, 12].

Table 5. Transition relation and termination-upon-transition relation for μCRL_t , where $a \in \mathcal{A}$, $a, b, c \in \text{Act}$, $p, q, p', q' \in \mathcal{P}$, $H \subseteq \text{Act}$, $R: \text{Act} \rightarrow \text{Act}$, $d, e \in \mathcal{D}_s$ and $t \in T_T$.

$\frac{p \xrightarrow{a(d)}_t \checkmark \quad q \xrightarrow{b(d)}_t \checkmark \quad \gamma(a, b) = c}{p \parallel q \xrightarrow{c(d)}_t \checkmark}$		$\frac{p \xrightarrow{a(d)}_t \checkmark \quad q \xrightarrow{b(d)}_t \checkmark \quad \gamma(a, b) = c}{p q \xrightarrow{c(d)}_t \checkmark}$	
$\frac{p \xrightarrow{a}_t \checkmark \quad a \notin \text{dom}(R)}{\rho_R(p) \xrightarrow{a}_t \checkmark}$		$\frac{p \xrightarrow{a}_t \checkmark \quad a \in \text{dom}(R)}{\rho_R(p) \xrightarrow{R(a)}_t \checkmark}$	
$\frac{p \xrightarrow{a}_t \checkmark \quad \mathcal{U}_t(q)}{p \ll q \xrightarrow{a}_t \checkmark}$	$\frac{p \xrightarrow{a}_t \checkmark \quad a \notin H}{\partial_H(p) \xrightarrow{a}_t \checkmark}$	$\frac{p \xrightarrow{a}_t p' \quad \mathcal{U}_t(q)}{p \ll q \xrightarrow{a}_t p'}$	
$\frac{p \xrightarrow{a}_t \checkmark \quad \mathcal{U}_t(q)}{p \parallel q \xrightarrow{a}_t t \gg q}$	$\frac{q \xrightarrow{a}_t \checkmark \quad \mathcal{U}_t(p)}{p \parallel q \xrightarrow{a}_t t \gg p}$	$\frac{p \xrightarrow{a}_t \checkmark \quad \mathcal{U}_t(q)}{p \ll q \xrightarrow{a}_t t \gg q}$	
$\frac{p \xrightarrow{a}_t p' \quad \mathcal{U}_t(q)}{p \parallel q \xrightarrow{a}_t p' t \gg q}$	$\frac{q \xrightarrow{a}_t q' \quad \mathcal{U}_t(p)}{p \parallel q \xrightarrow{a}_t t \gg p q'}$	$\frac{p \xrightarrow{a}_t p' \quad \mathcal{U}_t(q)}{p \ll q \xrightarrow{a}_t p' t \gg q}$	
$\frac{p \xrightarrow{a(d)}_t p' \quad a \notin H}{\partial_H(p) \xrightarrow{a(d)}_t \partial_H(p')}$	$\frac{p \xrightarrow{a(d)}_t p' \quad a \notin \text{dom}(R)}{\rho_R(p) \xrightarrow{a(d)}_t \rho_R(p')}$	$\frac{p \xrightarrow{a(d)}_t p' \quad a \in \text{dom}(R)}{\rho_R(p) \xrightarrow{R(a)(d)}_t \rho_R(p')}$	
$\frac{p \xrightarrow{a(d)}_t \checkmark \quad q \xrightarrow{b(d)}_t q' \quad \gamma(a, b) = c}{p \parallel q \xrightarrow{c(d)}_t q'}$		$\frac{p \xrightarrow{a(d)}_t p' \quad q \xrightarrow{b(d)}_t \checkmark \quad \gamma(a, b) = c}{p \parallel q \xrightarrow{c(d)}_t p'}$	
$\frac{p \xrightarrow{a(d)}_t p' \quad q \xrightarrow{b(d)}_t q' \quad \gamma(a, b) = c}{p \parallel q \xrightarrow{c(d)}_t p' q'}$		$\frac{p \xrightarrow{a(d)}_t \checkmark \quad q \xrightarrow{b(d)}_t q' \quad \gamma(a, b) = c}{p q \xrightarrow{c(d)}_t q'}$	
$\frac{p \xrightarrow{a(d)}_t p' \quad q \xrightarrow{b(d)}_t \checkmark \quad \gamma(a, b) = c}{p q \xrightarrow{c(d)}_t p'}$		$\frac{p \xrightarrow{a(d)}_t p' \quad q \xrightarrow{b(d)}_t q' \quad \gamma(a, b) = c}{p q \xrightarrow{c(d)}_t p' q'}$	

Table 6. Ultimate delay operator for μCRL_t , where $p, q \in \mathcal{P}$, $H \subseteq \text{Act}$, $R: \text{Act} \rightarrow \text{Act}$ and $t \in T_T$.

$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(p \parallel q)}$	$\frac{\mathcal{U}_t(q)}{\mathcal{U}_t(p \ll q)}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(p \parallel q)}$	$\frac{\mathcal{U}_t(q)}{\mathcal{U}_t(p \ll q)}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(p \parallel q)}$	$\frac{\mathcal{U}_t(q)}{\mathcal{U}_t(p \ll q)}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(p \parallel q)}$	$\frac{\mathcal{U}_t(q)}{\mathcal{U}_t(p \ll q)}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(\partial_H(p))}$	$\frac{\mathcal{U}_t(p)}{\mathcal{U}_t(\rho_R(p))}$

An Optimal Approach to Hardware/Software Partitioning for Synchronous Model

Pu Geguang^{1,3*}, Dang Van Hung¹, He Jifeng^{1**}, and Wang Yi²

¹ International Institute for Software Technology
{ggpu,dvh,jifeng}@iist.unu.edu
United Nations University, Macau.

² Department of Computer Systems, Uppsala University, Sweden.

³ LMAM, Department of Informatics, School of Math.
Peking University, Beijing, China 100871
yi@docs.uu.se

Abstract. Computer aided hardware/software partitioning is one of the key challenges in hardware/software co-design. This paper describes a new approach to hardware/software partitioning for synchronous communication model. We transform the partitioning into a reachability problem of timed automata. By means of an optimal reachability algorithm, an optimal solution can be obtained in terms of limited resources in hardware. To relax the initial condition of the partitioning for optimization, two algorithms are designed to explore the dependency relations among processes in the sequential specification. Moreover, we propose a scheduling algorithm to improve the synchronous communication efficiency further after partitioning stage. Some experiments are conducted with model checker UPPAAL to show our approach is both effective and efficient.

Keywords: Hardware/software partitioning, timed automata, reachability, scheduling algorithm.

1 Introduction

Computer system specification is usually completely implemented as software solution. However, some strong requirements for performance of the system demand an implementation fully in hardware. Consequently, in between the two extremes, Hardware/software Co-design [25], which studies systematically the design of systems containing both hardware/software components, has emerged as an important field. A critical phase of co-design process is to partition a specification into hardware and software components.

* Partially supported by NNSFC No. 60173003

** On leave from East China Normal University. The work is partially supported by the 973 project 2002CB312001 of the ministry of science and technology, and the 211 project of the ministry of Education of China.

One of the objectives of hardware/software partitioning is to search a reasonable composition of hardware and software components which not only satisfies the constraints such as timing, but also optimized desired quality metrics, such as communication cost, power consumption and so on.

Several approaches based on algorithms have been developed, as described, for example, in [2,20,21,24,26]. All these approaches emphasize the algorithmic aspects, for instance, integer programming [20,26], evolution algorithm [24] and simulated annealing algorithm [21] are respectively introduced to the partitioning process in the previous researches. These approaches are applied to different architectures and cost functions. For example, in [26], Markus provided a technique based on integer programming to minimize the communication cost and total execution time in hardware with certain physical architecture. The common feature of these approaches is that the communication cost is simplified as a linear function on data transfer or the relation between adjacent nodes in task graph. This assumption is reasonable in asynchronous communication model, but is unreasonable in synchronous communication model in which the cost of waiting time for communication between processes is very high. In order to manage the synchronous model which is common in many practical systems, a new approach should be introduced into the partitioning problem.

A number of papers in the literature have introduced formal methods into the partitioning process [17,23,3]. Some of them adopted a subset of the Occam language [16] as specification language [17,23]. In [23], for example, Qin provided a formal strategy for carrying out the partitioning phase automatically, and presented a set of proved algebraic laws of the partitioning process. In that paper, he did not deal with the optimization of the partitioning.

Few approaches deal with the analysis of the specification for exploring the hidden concurrency to relax the initial condition of the partitioning for optimization. In [17], Iyoda *et al.* provided several algebraic laws to transform the initial description of the system into a parallel composition of a number of simple processes. However, the method delivers a large number of processes and communication channels, which not only poses a complicated problem of merging those small processes, but also raises the communication load between the hardware and software components.

In this paper, we present an optimal automatic partitioning model. In this model, we adopt an abstract synchronous architecture composed of a coprocessor board and a hardware device such as FPGAs, ASIC etc, where the communication between them is synchronized. By means of our approach, the following goals will be achieved

- Explore the hidden concurrency, i.e, find the processes which could be executed in parallel from the initial sequential specification.
- Obtain the optimal performance of the overall program in terms of the limited resources in hardware. The communication waiting time between software and hardware components is considered as well.
- Improve the communication efficiency by re-ordering the commands to reduce the communication waiting time between hardware and software components after partitioning.

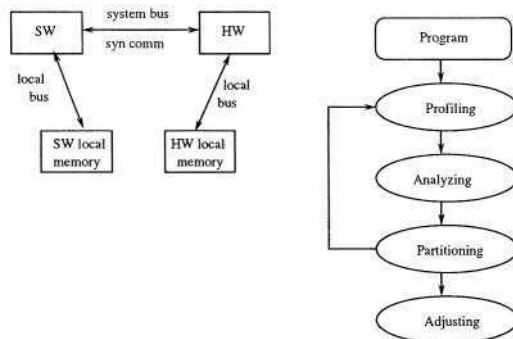


Fig. 1. Architecture and Partitioning Flow

Given a specification, system designers are required to divide the specification into a set of basic processes (blocks) which are regarded as candidate processes for the partitioning phase. In general, the next step is to select and put some processes into hardware components to obtain the best performance. On account of the parallel structure of software and hardware components, the hidden concurrency among the processes will relax precedence condition of the partitioning, that is, an optimal solution will be obtained from a larger search space. We design two algorithms to explore the control and data flow dependency. To allocate the processes into the software and hardware components, we transform the partitioning to a reachability problem of timed automata [10], and obtain the best solution by means of an optimal reachability algorithm. Since the synchronous communication model is adopted in our target architecture, to reduce the communication waiting time further, we adjust communication commands in the program of each component by applying a scheduling algorithm.

The paper is organized as follows. Section 2 presents the overview of our technique. Section 3 explores the dependency relation between processes. Section 4 describes a formal model of hardware/software partitioning using timed automata. In Section 5, we propose a scheduling algorithm to improve the communication efficiency. Some partitioning experiments are conducted in Section 6. Finally, Section 7 is a short summary of the paper.

2 Overview of Our Partitioning Approach

In this section we present the overview of our approach to hardware/software partitioning problem. The partitioning flow is depicted as Figure 1.

In profiling stage, a system specification is divided into a set of basic candidate processes which could never be split further. However, there is a trade-off between the granularity of the candidates and the feasibility of optimization. The smaller the candidate processes are, the greater the number of different partitions is. The large number of partitions will increase the time dramatically in the computation of the optimum. Furthermore, smaller candidate processes

will bring heavy communication cost. On the other hand, larger candidates will restrict the space of possibilities, therefore, may reduce the concurrency and increase the waiting time for communication. We leave this choice to the designers to repeat the profiling process as long as the partitioning results are not satisfied with according to the current granularity of candidate process. Once the designer decides the granularity, the initial specification is transformed into a sequential composition of candidate processes, that is, $P_1; P_2; \dots; P_n$, where P_i denotes the i th process.

The analyzing phase is to explore the control and data flow dependency among the sequential processes $P_1; P_2; \dots; P_n$. The data flow dependency is as important as the control flow dependency and helps to decide whether data transfer occurs between any two processes. The details are discussed in Section 3.

Our goal is to select those processes which yield the highest speedup if moved to hardware. More precisely, the total execution time is minimized in terms of the limited resources in hardware. The overhead of the required communication between the software and hardware should be included too. The synchronous waiting time will be considered in the performance of the partitioning as well. In fact, we will see that this partitioning is a scheduling problem constrained by precedence relation, synchronous communication and limited resources. We transform the scheduling problem into a reachability one of timed automata(TA) [10] and obtain the optimal result using an optimal reachability algorithm. TA is the finite state automata with clock variables. It is a useful formalism to model real-time systems [18], where the system verification is usually converted to checking of reachability properties, i.e. if a certain state is reachable or not. Automatic model checking tools for timed automata are available, such as UPPAAL [19], KRONOS [7] and HyTech [13]. We will use the UPPAAL as our modelling tool to conduct some partitioning experiments in Section 7.

When the partitioning process is finished, we get the following form:

$$P_{S_1}; P_{S_2}; \dots; P_{S_m} \parallel P_{H_1}; P_{H_2}; \dots; P_{H_l}$$

where P_{S_i} ($1 \leq i \leq m$) denotes a process allocated in the software component, and P_{H_i} ($1 \leq i \leq n$) denotes a process allocated in the hardware component.

In the end of partitioning phase, communication commands are added to support data exchange between the software and hardware. To reduce the waiting time, we reorganize the software and hardware components by re-ordering the communication commands. For example, let us consider the following partitioned processes $P \parallel Q$.

$P :$	$Q :$
$P1(x);$	$Q1(z);$
$y := f(x);$	$q := g(z);$
$P2(x);$	$Q2(z);$
$C!y;$	$C?q;$
$P3(x);$	$Q3(z);$

Suppose process P is implemented in software and Q is implemented in hardware. Where $C!y$ denotes the output and $C?q$ as the input. In the process P , moving the action $C!y$ before $P2(x)$ or after $P3(x)$, and in the process Q moving the action $C!q$ before $Q2(x)$ or after $Q3(x)$ will not effect the result of the program $P||Q$. We assume that the estimate of the execution time of P_1 , P_2 and P_3 is 2, 2, and 2 respectively, and the estimate of the time for the execution of Q_1 , Q_2 and Q_3 is 1, 1 and 1 respectively. Then moving $C!y$ to the line in between $y := f(x)$ and $P2(x)$ will make the program run faster. We propose a general algorithm which is applicable to more than two parallel processes to improve the communication efficiency.

3 Exploring Dependency Relations between Processes

3.1 Dependency Relations

Let $P_1; P_2; \dots; P_n$ be the initial sequential specification produced by the system designer in the profiling stage. In this section we explore the dependency relations between any two processes. This is an important step in analyzing phase. Our intention is to disclose the control and data flow relations of processes in the specification. These relations will be passed to the next step as the input for partitioning using timed automata model. Moreover, through the analysis of control relation among processes, we will find those processes that are independent so that they can be executed in any order on one component or in parallel on two components without any change to the computation result specified by the original specification.

Let for process P_i , $Wr(P_i)$ and $Re(P_i)$, respectively, denote the set of variables modified by P_i and the set of variables read by P_i .

The control flow dependency is represented by the relation $\prec$ over two processes defined as follows.

Definition 1

$$P_i \prec P_j =_{def} \begin{aligned} & (Wr(P_i) \cap Re(P_j) \neq \emptyset \vee \\ & Re(P_i) \cap Wr(P_j) \neq \emptyset \vee \\ & Wr(P_i) \cap Wr(P_j) \neq \emptyset) \wedge \\ & i < j \end{aligned}$$

We call P_i as a control predecessor of P_j . If $P_i \prec P_{i+1}$ does hold, then process P_{i+1} can not start before the completion of P_i . Otherwise, P_{i+1} can be activated before P_i leaving the behaviour of the whole program unchanged.

Theorem 1

$$P_i; P_{i+1} = P_{i+1}; P_i \text{ if } \neg(P_i \prec P_{i+1})$$

Proof. To prove formally this property we follow the convention of Hoare and He [14] that every program can be represented as a *design*. A design has the form $pre \vdash post$, where *pre* denotes the precondition and *post* denotes the post-condition. Sequential composition is formally defined as follows [14]:

$\mathbf{P}(v, v'); \mathbf{Q}(v, v') =_{def} \exists m \bullet \mathbf{P}(v, m) \wedge \mathbf{Q}(m, v')$,

Where variable lists v and v' stand for initial and final values respectively, and m is a set of fresh variables which denote the hidden observation.

The following lemma is taken from [14]:

Lemma 1

$$(\mathbf{P}_1 \vdash \mathbf{Q}_1); (\mathbf{P}_2 \vdash \mathbf{Q}_2) = \mathbf{P}_1 \wedge \neg(\mathbf{Q}_1; \neg\mathbf{P}_2) \vdash \mathbf{Q}_1; \mathbf{Q}_2$$

Because processes P_i and P_{i+1} do not satisfy relation $P_i \prec P_j$, we can easily obtain $Wr(P_i) \cap (Wr(P_{i+1}) \cup Re(P_{i+1})) = \emptyset$ and $Wr(P_{i+1}) \cap (Wr(P_i) \cup Re(P_i)) = \emptyset$. For simplicity, Assume $Wr(P_i) = \{y\}$, $Wr(P_{i+1}) = \{z\}$, $Re(P_i) \cap Re(P_{i+1}) = \{x\}$, where variables x, y, z stand for list variables respectively. Let $P_i = pre_1 \vdash post_1$, $P_{i+1} = pre_2 \vdash post_2$. We could note $pre_1, post_1, pre_2$ and $post_2$ as follows:

$$\begin{aligned} Pr_1 &= pre_1(x, y) \\ Pr_2 &= pre_2(x, z) \\ Po_1 &= post_1(x, y, y') \wedge x = x' \wedge z = z' \\ Po_2 &= post_2(x, z, z') \wedge x = x' \wedge y = y' \end{aligned}$$

From the above, we could easily obtain:

$$P_i = pre_1 \vdash post_1 = Pr_1 \vdash Po_1$$

$$P_{i+1} = pre_2 \vdash post_2 = Pr_2 \vdash Po_2$$

$$\begin{aligned} (1) \quad & Po_1; Po_2 && \{\text{rewrite } Po_1 \text{ and } Po_2\} \\ &= post_1(x, y, y') \wedge x = x' \wedge z = z' ; && \\ & \quad post_2(x, z, z') \wedge x = x' \wedge y = y' && \{\text{def of } ;\} \\ &= \exists m_1, m_2, m_3 \bullet post_1(x, y, m_2) \wedge && \\ & \quad x = m_1 \wedge z = m_3 \wedge post_2(m_1, m_3, z') \wedge && \\ & \quad m_1 = x' \wedge m_2 = y' && \{\text{predicate calculus}\} \\ &= \exists m_1, m_2, m_3 \bullet post_1(x, y, m_2) \wedge && \\ & \quad m_1 = x' \wedge m_2 = y' \wedge post_2(m_1, m_3, z') \wedge && \\ & \quad x = m_1 \wedge z = m_3 && \{\text{predicate calculus}\} \\ &= post_1(x, y, y') \wedge post_2(x, z, z') \wedge x = x' && \\ (2) \quad & Po_2; Po_1 && \{\text{rewrite } Po_1 \text{ and } Po_2\} \\ &= post_2(x, z, z') \wedge x = x' \wedge y = y' ; && \\ & \quad post_1(x, y, y') \wedge x = x' \wedge z = z' && \{\text{def of } ;\} \\ &= \exists m_1, m_2, m_3 \bullet post_2(x, z, m_3) \wedge && \\ & \quad x = m_1 \wedge y = m_2 \wedge post_1(m_1, m_2, y') \wedge && \\ & \quad m_1 = x' \wedge m_3 = z' && \{\text{predicate calculus}\} \\ &= \exists m_1, m_2, m_3 \bullet post_2(x, z, m_3) \wedge && \\ & \quad m_1 = x' \wedge m_3 = z' \wedge post_1(m_1, m_2, y') \wedge && \\ & \quad x = m_1 \wedge y = m_2 && \{\text{predicate calculus}\} \\ &= post_2(x, z, z') \wedge post_1(x, y, y') \wedge x = x' && \end{aligned}$$

(3) From (1)(2), we establish

$$Po_1; Po_2 = Po_2; Po_1$$

In the same way, we can prove the following equation

$$Pr_1 \wedge \neg(Po_1; \neg Pr_2) = Pr_2 \wedge \neg(Po_2; \neg Pr_1)$$

From Lemma 1, the theorem is proved. $\square$

Let set S_j ($1 \leq j \leq n$) store all the control predecessors of P_j , and constant max_j be the maximum index of processes in S_j . To uncover the hidden concurrency among the processes, we have the following corollary.

Corollary 1

$$\begin{aligned} &P_1; \dots; P_{max_j}; P_{max_j+1}; \dots; P_j; \dots; P_n \\ &= P_1; \dots; P_{max_j}; P_j; P_{max_j+1}; \dots; \dots; P_n \\ &\text{if } max_j < j - 1 \end{aligned}$$

Proof. Apply Theorem 1 ($j - 1 - max_j$) times. $\square$

This corollary shows each process P_k between P_{max_j} and P_j could be executed in parallel with P_j . If P_k and P_j are allocated in the software and hardware respectively, it should reduce the execution time of the whole program.

To be more concrete on the data flow specified by the initial specification, we introduce the relation $\overset{d}{\prec}$ between processes which is exactly the relation “read-from” in the theory of concurrency control in databases.

Definition 2

$$\begin{aligned} P_i \overset{d}{\prec} P_j &= i < j \wedge \exists x \bullet (x \in Wr(P_i) \cap Re(P_j) \wedge \\ &\quad \forall k \bullet (i < k < j \Rightarrow x \notin Wr(P_k))) \end{aligned}$$

If processes P_i and P_j satisfy relation $\overset{d}{\prec}$, there is direct data transfer between them in any execution. We call P_i as a data predecessor of P_j . Through this relation, we know the data from which process, a process may need and estimate the communication time between them.

3.2 Algorithms for Exploring Dependency Relations

In this section, we present two algorithms. One is for finding control predecessors of each process, and the other is for finding data predecessors of each process. The two algorithms are intuitive, so we will omit the proof of their correctness here.

Set variables S and T are vectors with n components, and store the control and data predecessors of each process respectively. i.e, the postconditions of the two algorithms are as follows,

$$\forall j \leq n \bullet (S_j = \{P_k \mid P_k \prec P_j\})$$

Table 1. Control and Data Flow Dependency Algorithms

Input: $P_1; P_2; \dots P_n$ Output: <i>Set vector S</i> Method: for $j := 1$ to n $S_j := \emptyset;$ end for for $j := 2$ to n for $i := 1$ to $j - 1$ if $(P_i \prec P_j)$ $S_j := S_j \cup \{P_i\};$ endif endfor endfor	Input : $P_1; P_2; \dots P_n$ Output : <i>Set vector T</i> Method: for $i := n$ to 1 $T_i := \emptyset;$ end for for $i := n$ to 1 $j := i - 1;$ while $(Re(P_i) \neq \emptyset \wedge j > 0)$ $G := Re(P_i) \cap Wr(P_j);$ if $(G \neq \emptyset)$ $T_i := T_i \cup \{P_j\};$ $Re(P_i) := Re(P_i) \setminus G$ end if $j := j - 1;$ end while end for

$$\forall i \leq n \bullet (T_i = \{P_k \mid P_k \stackrel{d}{\prec} P_i\})$$

Obviously, $S_1 = T_1 = \emptyset$. Table 1 shows the two algorithms.

Although the control dependency algorithm is very simple, the set S_j ($1 \leq j \leq n$) discloses the hidden concurrency in the sequential specification based on the corollary in the last subsection.

The set vector variables S and T will provide us all necessary information on the temporal order between processes which will be the input for modelling the partitioning with timed automata in UPPAAL. For simplicity, let D_i be the set of indexes for the processes in T_i , i.e. $D_i = \{j \mid P_j \in T_i\}$

4 Formal Model Construction

In this section, we transform the hardware/software partitioning into a reachability problem of timed automata. The timed behaviour of each process is modelled as a timed automaton, and the whole system is composed as a network of timed automata. The timed automata models of all processes are similar except for some guard conditions. After the model is constructed, an optimal reachability algorithm is applied to obtain an optimal trace in which each process is sequentially recorded whether it is allocated in hardware or software components. As model checker UPPAAL has implemented this algorithm in its latest version, we use the UPPAAL as our modelling tool.

4.1 Behaviour Analysis

Here we list some key elements of the timed automata in our model.

Table 2. Notation

P_i	Process P_i
$Token_i$	The number of control predecessors of P_i
$Tcom_i$	The estimated communication time of P_i
$STexe_i$	The estimated execution time of P_i if implemented in software
$HTexe_i$	The estimated execution time of P_i if implemented in hardware
Rh_i	The estimated hardware resources needed for P_i
St_i	Variable indicating the state of P_i
X_i	Variable recording the number of the terminated control predecessors of P_i
SR	Variable indicating if the processor of software is occupied
HR	Variable indicating if the hardware is occupied
$Hres$	Constant storing the total available hardware resources
CS_i	Software clock for P_i
CH_i	Hardware clock for P_i
CC_i	Communication clock for P_i
T_i	The set of the processes from which P_i reads data from
D_i	The set of the indexes of the processes that P_i reads data from

State variables. Each process has two possible states which indicate whether the process is allocated in hardware or software. We use a global variable St_i to record the state of P_i . St_i is 1 if process P_i is implemented in software, otherwise it is 0.

Precedence constraints. It is obvious that only when all the control predecessors of a process have terminated, the process has the opportunity to be executed either in hardware or software. We use local variable X_i to denote the number of the control predecessors of P_i which have completed their tasks.

Resource constraints. There is only one general processor in our architecture, so no more than one process can be executed on the processor at any time. We introduce a global variable SR to indicate whether the processor is occupied or not. The processor is free if SR is 1, otherwise SR is 0. As far as hardware resources are concerned, the situation is a little complicated. We introduce global variable $Hres$ to record the available resources in hardware. As the processes in hardware are also sequential like in software in our architecture, we introduce a global variable HR to denote whether a process is executed in hardware. If HR is 1, it indicates that no process occupies the hardware. Otherwise HR is 0.

Clock variables. Local clock variables CH_i and CS_i represent the hardware clock and software clock for process P_i respectively. To calculate the communication time between the software and hardware we introduce local clock CC_i for process P_i .

Table 2 lists the variables used in our timed automata model together with their intended meaning. Most of these notations have been explained above.

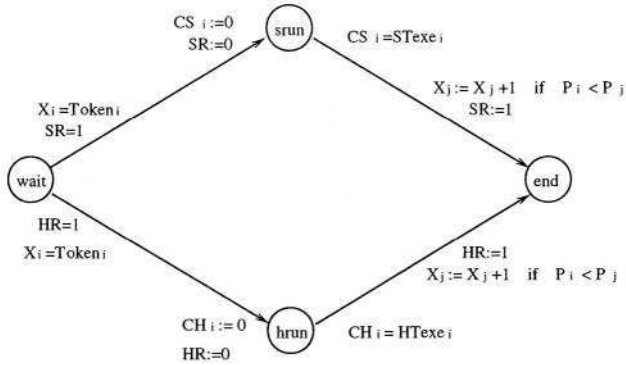


Fig. 2. The Simple Model of Process P_i

4.2 Model Construction

In this section we present two models, one is called simple model and helps to understand the behaviour of the system, and the other is the full model which takes into account all the elements including resource and communication.

Simple Model. Figure 2 shows the simple model. It expresses the timed behaviour of process P_i . There are four states in this model. The *wait* state denotes that process P_i is waiting. The states *srun* and *hrun* denote that P_i is allocated in software or in hardware respectively. When P_i finishes its computation task, it will be in the state *end*. Our purpose is to find the fastest system trace in which all processes reach their *end* states.

If the control predecessors of process P_i are all terminated, i.e. $X_i = \text{Token}_i$ is satisfied, P_i is enabled to be executed in hardware or software. When both of the components are free, it will choose one of them nondeterministically. If both components are occupied by other processes, P_i will still be in state *wait*.

Suppose process P_i is decided to run in software. Once P_i has occupied the software, it sets the global variable SR to 0 to prevent other processes from occupying the processor. It sets the software clock CS_i to 0 as well. The transition between the *srun* state and *end* state can only be taken when the value of the clock CS_i equals STexe_i . As soon as the transition is taken, variable X_j will be added one if P_i is the control predecessor of process P_j . At the same time, P_i releases the software processor as well. The situation is similar if P_i is implemented in the hardware.

This simple model shows that every P_i may be implemented in software or in hardware. When all the processes reach its *end* state, the reachability property of the system is satisfied. To obtain the minimal execution time of the whole system, we use a global clock in UPPAAL tool. When the optimal reachability trace is found the global clock will show the minimal execution time.

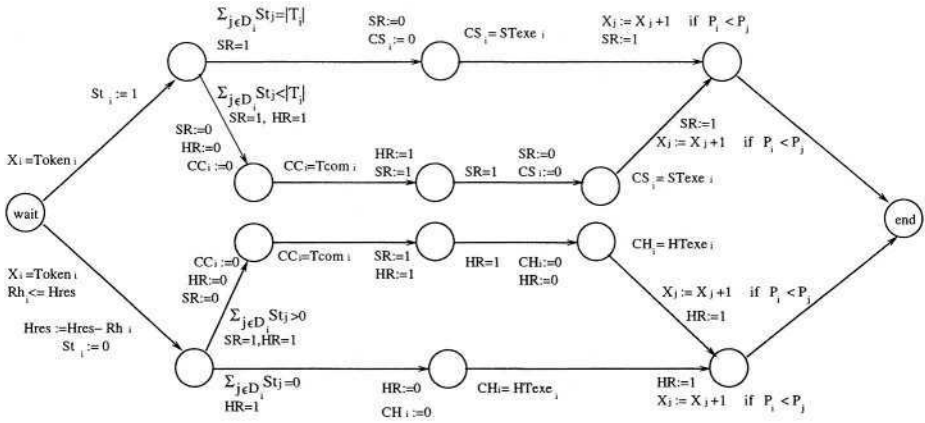


Fig. 3. The Full Model of Process P_i

Full Model. Now we present the full model taking into account the communication and resource, etc. The full model is depicted in Figure 3.

In addition to the states the simple model introduced, we should solve two problems which are not involved before. One is how to simulate the resource allocation in the hardware component, and the other is how to simulate the data transfer between the hardware and software.

The first problem is simple. When considering allocation of process P_i in hardware, the automata tests not only variable HR but also variable $Hres$ to check if the hardware resources are sufficient for P_i . If the resources there are enough, P_i may be put into the hardware. Though there exist reusable resources in hardware, such as *adder* and *comparator*, we do not need to consider them here because the processes are implemented sequentially in hardware in our target architecture. When process P_i terminates, it releases the reusable resources for other processes.

In order to model the communication between the software and hardware, the data dependency among processes has to be considered. When process P_i uses the data which is produced by other processes, there is data transfer between them. If they are all in the same component, the communication time could be ignored. For example, when the processes communicated with each other are all in software, they exchange data via shared memory. Supposing that process P_i is in the software, and at least one process communicating with P_i is allocated in the hardware, the communication between them will take place by means of the bus or other physical facilities. In this case, the overhead of communication between the software and hardware can not be negligible, and should be taken it into account in the model.

Recall that variable St_i is introduced to denote that process P_i is implemented in the hardware or software components. For example, when P_i is allocated in software, St_i is set to 1. P_i then checks the processes that will transfer

data to it (i.e. the processes in T_i) are in software or hardware. If at least one of them is in the hardware, the communication should be taken into account. In Figure 3, the condition $\sum_{j \in D_i} St_j < |T_i|$ is a guard to express that at least one process that P_i reads data from is in hardware. The situation is the same when P_i is located in hardware and $\sum_{j \in D_i} St_j > 0$.

Next, when the communication really occurs between the software and hardware, it should occupy both the software and hardware components. That is to say, no other processes would be performed until the communication is finished. According to this, the variables SR , HR are set to 0 simultaneously as long as the communication takes place. The clock CC_i is set to 0 when the communication begins. For the communication time T_{comm_i} depends on the states of P_i 's data predecessors, there are two methods to estimate the value of T_{comm_i} . One is to adopt T_{comm_i} as the probability average value of all the communication combinations. On the other hand, we can calculate the values of all possible communication in advance, then choose T_{comm_i} as one of them according to the current states of process P_i 's data predecessors.

Once a communication action of process P_i finishes, the control of the hardware and software is released immediately. Process P_i will compete hardware or software resources with other ready processes at that point.

It is worthwhile to point out that even if process P_k is one of the data predecessors of P_j , it is not necessary that there will be a non negligible time consuming communication between processes P_j and P_k . Other process P_i may be as a delegate to transfer the data for them. The data will not be modified by process P_i in terms of the data dependency defined before. For example, if both processes P_i and P_j are implemented in the hardware and they have to transfer data to the process P_k which is allocated in the software. Process P_i or P_j will be a delegate to send all the data to P_k . Although more than one process will communicate with process P_k , the communication between the hardware and software occurs only once.

4.3 Discussion

We have showed that the hardware/software partitioning is formulated as a scheduling problem, which is constrained by precedence relation, limited resources, etc. In the partitioning model, we need not only to check all the processes could reach their *end* states, but also to obtain a shortest accumulated delay trace. This is regarded as the optimal reachability problem of model checking in timed automata.

For model checking algorithm, it is necessary to translate the infinite state-spaces of timed automata into finite state presentation. For pure reachability analysis, symbolic states [11] of the form (l, Z) are often used, where l is a location of the timed automata and Z is a convex set of clock valuations called a *zone*. The formal definition of (l, Z) could be found in [11]. Several optimal reachability algorithms have been developed based on this presentation of symbolic state, such as [12,5,4].

To generalize the minimum-time reachability, in [5], a general model called *linearly priced timed automata* (LPTA), which extends the model of TA with *prices* on all transitions and locations is introduced to solve the minimum-cost reachability problem. *Uniformly Priced Timed Automata* (UPTA) [4], as a variety of LPTA, adopts a heuristic algorithm which uses some techniques such as branch-and-bound to improve the searching efficiency has been implemented in the latest version of UPPAAL. In Section 6, we will use UPPAAL to do some experiments on some hardware/software partitioning cases.

5 Improving the Communication Efficiency

After the partitioning stage is finished, we obtain two parallel processes running in software and hardware components respectively. The communication is synchronized between them. Moreover, we can improve communication efficiency further by moving the communication commands appropriately.

The idea is that we can find a flexible interval $[ASAP, ALAP]$ for each communication command in which the communication could occur without changing the semantics of the program. This interval denotes the *earliest* and *latest* time when the communication command can execute relatively to the computation time of a program. Then we apply a scheduling algorithm to decide the appropriate place of communication command to reduce the waiting time between processes. Here we propose a general algorithm which is for more than two processes in parallel.

5.1 System Modelling

Let S be a system of m processes $\{P_1, \dots, P_m\}$ running in parallel and synchronized by handshaking. All the processes start at time 0. In our partitioning problem, let m equal 2.

Description of Each Process P_i

- P_i has a communication trace $\alpha_i = c_1^i c_2^i \dots c_{n_i}^i$, $c_j^i \in \Sigma_i$, where Σ_i is the alphabet of the communication actions of P_i .
- P_i needs computation time A_i before it terminates.
- Each c_j^i has a “flexible” interval for the starting time $[a_j^i, b_j^i]$ relatively to the computation time of P_i , $a_j^i \leq b_j^i \leq A_i$. This means that c_j^i is enabled when the accumulated execution time of P_i has been reached a_j^i time units, and should take place before the accumulated execution time of P_i reaches b_j^i time units. b_j^i and A_i can be infinity, and a_j^i can be 0. To be meaningful, we assume that $a_j^i \leq a_{j+1}^i$ and $b_j^i \leq b_{j+1}^i$ for $1 \leq j < n_i$.
- P_i is either running or waiting when not yet completed. It is waiting iff it is executing a communication action c_j^i for which the co-action has not been executed.

We now formulate the problem precisely. The purpose of formalization here is just to avoid ambiguity and to simplify the long text in the proof when applicable. Any formalism to model the problem must have the capacity to express the “accumulated execution time” for processes. For this reason, we take some idea from Duration Calculus (DC) ([8]) in the formalization.

For each process P_i , we introduce four state variables (which are mappings from $[0, \infty)$ to $\{0, 1\}$) $P_i.running$, $P_i.waiting$, $P_i.completed$ and $P_i.start$ to express the states of P_i . At time t , the state variable $P_i.running$ ($P_i.waiting$, $P_i.completed$ and $P_i.start$) has the value 1 iff P is running (waiting, completed and start, correspondingly) at the time. Process P_i starts at time 0 and terminates when its accumulated execution time has reached A_i . All processes stay in the state “complete” when they terminate.

Systems and Assumptions

- $\alpha_1, \alpha_2, \dots, \alpha_m$ are assumed to be matched in the sense of handshaking synchronization. Let f be the matching function, i.e. $f(c_j^i, c_h^k) = 1$ iff c_j^i and c_h^k are matched (they are the partners for the same communication).
- Let t_j^i be starting time of c_j^i (according to the unique global clock). Then t_j^i must satisfy the constraint for c_j^i , i.e. $a_j^i \leq \int_0^{t_j^i} P_i.running.dt \leq b_j^i$ and $t_j^i \leq t_{j+1}^i$.
- $P_i.waiting(t) = 1$ if and only if there exists c_j^i and c_h^k such that $t_j^i \leq t \leq t_h^k$ and $f(c_j^i, c_h^k) = 1$. (P_i is waiting iff it decides to communicate and its partner is not ready).

To formalize the behaviour of communication actions as mentioned in the items above, we introduce for each c_j^i and c_h^k such that $f(c_j^i, c_h^k) = 1$, a state variable $comm(i, j, k, h)$. $comm(i, j, k, h)(t) = 1$ iff at time t , one of the partner action (either c_j^i or c_h^k) has started and the communication has not completed. Note that $comm(i, j, k, h) = comm(k, h, i, j)$.

An execution of S is a set of intervals $[t_j^i, t_{j'}^i]$ of the starting time and the ending time for communication actions c_j^i . An execution terminates at time t iff t is the termination time of the latest process.

Question: Develop a procedure for the scheduling to produce an execution of S that terminates at the earliest time.

In the following algorithm and example, we assume that the communication takes no time for simplicity. The algorithm is still correct when including the communication time.

5.2 Scheduling Algorithm

Because $\alpha_1, \dots, \alpha_m$ are matched, we can easily construct a dependency graph G to express the synchronised computation for the system S (a Mazurkiewicz trace [1], [9]) as follows. Each node of the graph represents a synchronised action (c_j^i, c_h^k) with $f((c_j^i, c_h^k)) = 1$ (and labelled by (c_j^i, c_h^k)). There exists a directed edge from $n_1 = (c_j^i, c_h^k)$ to $n_2 = (c_{j'}^{i'}, c_{h'}^{k'})$ iff either $i = i'$ and $j' = j + 1$ or $k = k'$ and $h' = h + 1$. G is used as an additional input for the algorithm.

Algorithm

Input: S, G

Output: Time to start each communication action for each process and time for the synchronous communication actions (represented by each node n of G)

Method:

- (1) (Initialisation) Set waiting time vector to zero $W := (0, 0, \dots, 0)$ (no process is waiting before doing any synchronisation action). Set the last communication time vector V to zero $V = (0, 0, \dots, 0)$.
- (2) Compute the minimal slice C for G which is the set of all nodes of G with no incoming edge. If $C = \emptyset$ halt the algorithm. Otherwise, for each node $n = (c_j^i, c_h^k)$ in C :
 - (2.1) Compute the global real time intervals for the enabling of c_j^i and c_h^k : $I := [W_i + a_j^i, W_i + b_j^i]$ and $J := [W_k + a_h^k, W_k + b_h^k]$. Let $K := [\max\{V_i, V_k\}, \infty)$ be the interval of possible time for the synchronous communication action represented by the node n .
 - (2.2) (Select the earliest time t_n that (c_j^i, c_h^k) can take place). Let $B = I \cap J \cap K$.
 - (a) If $B \neq \emptyset$ then $t_n := \min B$. In this case, $t_j^i := t_h^k := t_n$ (no waiting time), $V_i := V_k := t_n$.
 - (b) If $B = \emptyset$ then $t_n := \min I \cap K$ if $\max J \cap K < \min I \cap K$, and update the waiting time $W_k := W_k + (\min I \cap K - \max J \cap K)$. In this case, $t_j^i := t_n$, $t_h^k := \max J \cap K$, and $V_i := V_k := t_n$. The case $\max I \cap K < \min J \cap K$ is symmetric.
- (3) Remove all the nodes in C and the edges leaving from them from graph G .
- (4) Repeat Step 2 until G is empty.
- (5) Output t_j^i for each j, i , and t_n (as the scheduled time for the communication actions represented by the node n) for each node $n = (c_j^i, c_h^k)$ of G .

Example 1. Suppose there are 3 processes P_1, P_2 and P_3 to communicate each other. The communication intervals of the processes are showed in Fig 4. The dependency graph G for S is constructed as well.

The first execution of Step 2 is on the slice $C1 = \{n1\}$, and gives $t1 = 4$, $W = (0, 0, 0)$, $V = (4, 4, 0)$ meaning that until time $t1$ for the finishing of the actions represented by $n1$, no process is waiting, and that at the action represented by $n1$ involves $P1$ and $P2$, and terminate at time 4.

The second execution of Step 2 is on the slice $C2 = \{n2\}$, and gives $t2 = 6$, $W = (0, 0, 0)$, $V = (4, 6, 6)$ meaning that until time $t2$ for the finishing of the actions represented by $n2$, no process is waiting.

The last execution of Step 2 is on the slice $C3 = \{n3\}$, and gives $t3 = 11$, $W = (1, 0, 0)$, $V = (11, 11, 6)$ meaning that until time $t3$ for the finishing of the actions represented by $n3$, $P1$ has to wait for 1 time unit.

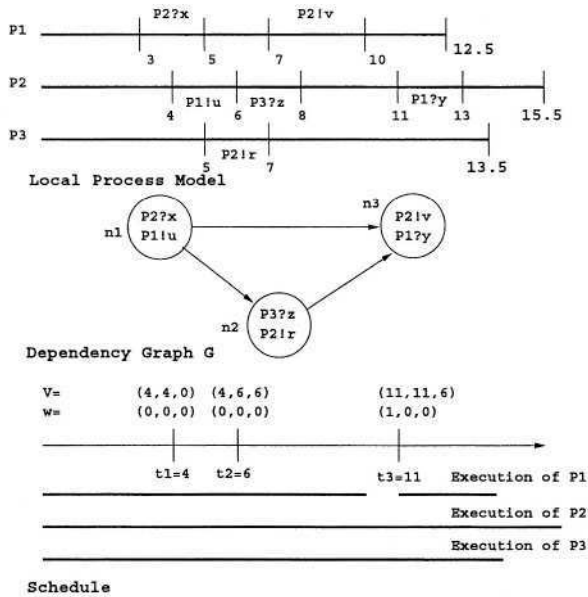


Fig. 4. The Algorithm Execution

Theorem 2. t_n is the earliest time that the communication actions represented by n can take place. Hence, the algorithm gives a correct answer to the problem mentioned above.

Proof. See [22]. □

6 Experiments in UPPAAL

We have used the technique in the previous section to find optimal solution for some hardware/software partitioning cases. In this section we present some of our experiments in solving them with the model checker UPPAAL version 3.3.32, running in Linux machine with 256Mb memory.

After we have modelled a hardware/software partitioning problem as a network of timed automata with n processes, we input the model to the UPPAAL model checker. Then we asked the UPPAAL to verify:

E <> P1.end and P2.end and ... Pn.end

This expression in UPPAAL specification language specifies that there exists a trace of the automata network in which eventually all n processes reach their *end* states.

To let UPPAAL find out the optimal solution to our problem, we choose the breadth-first model checking algorithm (UPPAAL offers various different al-

Table 3. Experimental results of partitioning using UPPAAL

Case studies	Num. of total processes	Required gates in hardware (K)	Num. of processes in hardware after partitioning	Running time in UPPAAL(Seconds)
Huffman decoder	10	44.5	5	121
rmatrix multiplier	20	97.7	3	289
packdata	30	221.9	5	503

gorithms) and the option “fastest trace” provided by UPPAAL. A global clock variable is declared to store the execution time. When the reachability property is satisfied, the fastest trace which records the partitioning scheme will be found, and the global clock will record the minimal execution time of all the processes. This trace, after having been added with the necessary communication statements, can be passed into the software compiler and hardware compiler to for implementing.

In the experiments, we use a Occam-like language as our specification language, and use the hardware compiler technique [6] to estimate the required resources in hardware of each process. For simplicity, as resources we list here only the estimated required gates of each problem.

The experimental results for the three case studies are shown in Table 3. We assume there are 15,000 gates in hardware resources. The second column of Table 3 shows the resources required if the program is entirely fulfilled in hardware for each case respectively.

The first one is Huffman decoder algorithm. The second is a matrix multiplier algorithm, and the last example is a pack data algorithm in network.

7 Summary

This paper presents a noval approach to hardware/software partitioning supporting the abstract architecture in which the communication takes place synchronously. After the designer decides the process granularity of the initial specification, the partitioning process could be carried out automatically. We explore the relations among processes to find the hidden concurrency and data dependency in the initial specification. These relations are as the input of timed automata to ensure the behaviours of processes are modelled correctly. Once the formal partitioning model is constructed with timed automata, the optimal result can be obtained by means of an optimal reachability algorithm. To further improve the synchronous communication efficiency between hardware and software components, a scheduling algorithm is introduced to adjust communication commands after partitioning. The experiments in model checker UPPAAL clearly demonstrated the feasibility and advantage of our proposed approach.

Acknowledgement. We are grateful to Liu Zhiming, Jin Naiyong for their helpful discussions and suggestions for the improvement of the paper.

References

1. I. J. Aalbersberg, G. Rozenberg. Theory of Traces. *Theoretical Computer Science*, Vol. 60, pp1-82, 1988.
2. S. Agrawal and R. Gupta. Dataflow-Assisted Behavioral Partitioning for Embedded Systems. *Proc. Design Automation Conf ACM, N.Y.* pp709-712, 1997.
3. E. Barros, W. Rosenstiel, X. Xiong. A Method for Partitioning UNITY Language in Hardware and Software. In *Proc. EURODAC*, September, pp220-225, 1994.
4. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, and J. Romijn. Efficient Guiding Towards Cost-Optimality in Uppaal. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, LNCS 2031, pp174-188, 2001.
5. G. Behrmann, A. Fehnker, T. Hune, K. G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-Cost Reachability for Priced Timed Automata. In *Proceedings of the 4th International Workshop on Hybrid Systems: Computation and Control (HSCC'01)*. LNCS 2034, pp147-161, 2001.
6. J. Bowen and He Jifeng. An approach to the specification and verification of a hardware compilation scheme. *Journal of Supercomputing*, 19(1):pp23-29, 2001.
7. M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine, "Kronos: A modelchecking tool for real-time systems," CAV'98, LNCS 1427, pp546-550, 1998.
8. Dang Van Hung. Real-time Systems Development with Duration Calculus: an Overview. Technical Report 255, UNU/IIST, P.O. Box 3058, Macau, June 2002.
9. V. Diekert and G. Rozenberg, editors. Book of Traces. World Scientific, Singapore, 1995.
10. R. Dill and D. L. Dill. A Theory for Timed Automata. In *Theoretical Computer Science*, Vol.125, pp183-235, 1994.
11. D. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Proc. of Automatic Verification Methods for Finite State Systems*, LNCS 407, pp197-212, 1989.
12. A. Fehnker. Bounding and heuristics in forward reachability algorithms. Technical Report CSI-R0002, Computing Science Institute Nijmegen, 2000.
13. Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HyTech: A Model Checker for Hybrid Systems. In *Proc. of the 9th Int. Conf. on Computer Aided Verification (Orna Grumberg, ed.)*, LNCS1254, pp460-463, 1997.
14. C. A. R. Hoare and He Jifeng. Unifying Theories of Programming. Prentice Hall, 1998.
15. T. Hune, K. G. Larsen, and P. Pettersson. Guided Synthesis of Control Programs Using UPPAAL. *Proc. of Workshop on verification and Control of Hybrid Systems III*, ppE15-E22, 2000.
16. INMOS Ltd. The Occam 2 Programming Manual. Prentice-Hall, 1988.
17. J. Iyoda, A. Sampaio, and L. Silva. ParTS: A Partitioning Transformation System. In *World Congress on Formal Methods 1999 (WCFM 99)*, pp1400-1419, 1999.
18. K. G. Larsen, P. Pettersson and Wang Yi. Model-Checking for Real-Time Systems. In *Proceedings of the 10th International Conference on Fundamentals of Computation Theory*, LNCS 965, pp62-88, 1995.
19. K. G. Larsen, P. Pettersson and Wang Yi. UPPAAL in a Nutshell. *Int. Journal of Software Tools for Technology Transfer* 1,1-2(Oct), pp134-152, 1997.
20. R. Nieman and P. Marwedel. An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming. *Design Automation for Embedded Systems, special Issue : Partitioning Methods for Embedded Systems*. Vol. 2, No. 2, pp165-193, Kluwer Academic Publishers, March 1997.

21. Z. Peng, K. Kuchcinski. An Algorithm for Partitioning of Application Specific System. *IEEE/ACM Proc. of The European Conference on Design Automation (EuroDAC)*, pp316-321, 1993.
22. Pu Geguang, Wang Yi, Dang Van Hung, and He Jifeng. An Optimal Approach to Hardware/software Partitioning for Synchronous Model. Technical Report 286, UNU/IIST, P.O. Box 3058, Macau, September, 2003.
23. Qin Shengchao and He Jifeng. An Algebraic Approach to Hardware/software Partitioning. *Proc. of the 7th IEEE International Conference on Electronics, Circuits and Systems*, (ICECS 2000), pp273-276, 2000.
24. G. Quan, X. Hu and G. W. Greenwood. Preference-driven hierarchical hardware/software partitioning. In *International conference on Computer Design(IEEE)*, pp652-657, 1999.
25. J. Staunstrup and W. Wolf, editors. *Hardware/Software Co-Design: Principles and Practice*. Kluwer Academic Publishers, 1997.
26. M. Weinhardt. Integer Programming for Partitioning in Software Oriented Code-sign. *Lecture Notes of Computer Science* 975, pp227-234, 1995.
27. W. Wolf. Hardware-Software Co-Design of Embedded System. *Proc. of the IEEE*, Vol.82, No.7, pp967-989, 1994.

A Many-Valued Logic with Imperative Semantics for Incremental Specification of Timed Models*

Ana Fernández Vilas, José J. Pazos Arias, Rebeca P. Díaz Redondo,
Alberto Gil Solla, and Jorge García Duque

Departamento de Ingeniería Telemática. Universidad de Vigo. 36200, Vigo, Spain
{avilas, Jose, rebeca, agil, jgd}@det.uvigo.es

Abstract. In order to reconcile the state of the art and the state of the practice in software engineering, immediate goals aim to use formal methods in ways that are minimally disruptive to professional practice. In this pursuit formal methods should be adapted to flexible lifecycle structures, getting over more traditional approaches. In the field of real-time design, SCTL/MUS-T methodology proposes a software process using formal methods that builds incrementally the model-oriented specification of the intended system. There are two main issues in this proposal: the incremental nature of the process, calling for a many-valued understanding of the world; and the construction of a model-oriented specification, calling for an imperative viewpoint in specification. From this starting point, this paper introduces a many-valued logic with imperative semantics enabling (1) to build a very-abstract level prototype from the scenarios identified on the intended system; (2) to capture the uncertainty and disagreement in an incremental process by providing a measure of how far or how close the prototype is from satisfying the intended requirements.

1 Introduction

Timed formal methods for the specification and analysis of real-time systems have been in the state of the art for a long time (see [1] for a survey in dense-time approaches¹). Despite being today a settled discipline, it is often thought that timed formal methods, and formal methods in general, run contrary to the state of the practice in software engineering. At this respect, iterative and incremental development is a major milestone for reconciling timed formal methods with current best practices in real-time design. Instead of looking at a set of features at once and producing a design for these features, in iterative and

* This work was partially supported by the Xunta de Galicia Basic Research Project PGIDT01PX132203PR.

¹ Dense-time approaches are unquestionably more expressive than discrete-time ones [2]. Despite this, dense-time formal methods have been revealed to be surprisingly tractable.

incremental development the software is designed and implemented feature by feature. Such flexible lifecycles are specially advisable in the real-time area where complexity calls for inspecting design alternatives, receiving immediate feedback and providing prototypes from the early phases of the process. At this respect, two main challenges have been pointed out in [3]:

- It should be enabled to start analysis much earlier, on incomplete specifications. This would ensure incremental gain for incremental effort.
- Methods should be provided for building correct specifications in a systematic and incremental way, so that high quality is guaranteed by construction.

In the pursuit of the above challenges, we propose a methodology, SCTL/MUS-T, which defines a software process using formal methods that builds iteratively and incrementally the specification of a real-time system. There are two main issues in this proposal: the incremental nature of the process calling for a many-valued understanding of the world; and the construction of an abstract prototype calling for an imperative (constructive) viewpoint in specification.

On one hand, we postulate that a smart approach to implicitly integrate incremental specifications in formal methods area can be provided by many-valued reasoning. In regard to formal methods, many-valued reasoning and specifically many-valued logics are known to support the explicit modeling of uncertainty and disagreement by allowing additional truth values outside bivalence principle [4]. In regard to the software process, in an incremental approach, the system gradually takes form as more is learned about the problem, that is, both uncertainty and disagreement are pervasive and desirable.

On the other hand, regarding the specification of real-time systems in terms of real-time logics, a variety of logics have been applied in the literature (see [5] for a survey). These real-time logics have mainly concentrated on the exploitation of its declarative features, so logic is taken as a language for describing properties of models in order to enable a posteriori analysis. Instead of describing what should be true, in imperative semantics the actions to be taken to ensure that it becomes true are described. So, a logical formula is not evaluated in a model but performs actions on that model to get a new one.

SCTL-T (Timed Simple Causal Temporal Logic) is a dense-time and many-valued logic for the incremental specification of real-time systems. SCTL-T is defined both (1) with a declarative semantics, for specifying the relevant requirements of the intended system and (2) with an imperative semantics, for incrementally building a model-oriented specification MUS-T (Timed Model of Unspecified States, section 2) by incorporating scenarios of the intended system. On the declarative side, SCTL-T requirements are evaluated in the incomplete MUS-T model of the system which enables to start analysis early (section 3.2). On the imperative side, SCTL-T^s scenarios perform actions in the incomplete model to provide the features in the scenario (section 3.4).

The declarative part of the methodology SCTL/MUS-T has been introduced in [6]. Besides, the methodology is demonstrated by a complete case study, the well known steam boiler, in [7]. This paper focuses on the formalization of the

imperative part of the methodology by means of SCTL-T^s scenarios. After introducing SCTL-T, the remainder of the paper is structured as follows. Section 4 describes the rationale behind the many-valued approach and the dynamics of the incremental process. Section 5 formalizes the iterative and incremental nature of the process by (1) establishing the relation among intermediate models and (2) proving the preservation of the knowledge when a new scenario is incorporated. Section 6 details some abstract examples. Then, some aspects of the implementation of a prototype tool based on the approach are discussed (section 7) and previous works close related to our approach are covered (section 8). Finally, section 9 summarizes main conclusions and discusses the topics we are working on.

2 MUS-T: Model for Incomplete Real-Time Systems

For modeling real-time systems we define MUS-T models (Timed Model of Unspecified States), which are actually a many-valued extension of the timed automata in [8]. Before defining MUS-T, some definitions concerning clocks and valuations are in order. A clock is a simple real-valued variable drawn from the finite set $\mathcal{C}$. A clock constraint $\psi \in \Psi(\mathcal{C})$ is a boolean combination of atomic formulas of the form $x \prec c$ or $x - y \prec c$, where x, y are clocks; c is an integer constant; and $\prec$ is taken from $\{\leq, \geq, <, >, =\}$. A clock valuation is a function $\gamma : \mathcal{C} \rightarrow \mathbb{R}_+$ assigning to each clock x a non-negative real number $\gamma(x)$. The set of all valuations is $\mathbb{R}_+^{\mathcal{C}}$. $\gamma + \tau$ for some $\tau \in \mathbb{R}_+$ stands for the clock valuation obtained by adding τ to the value of every clock. Given a clock constraint ψ , $\psi(\gamma) = 1$ iff γ satisfies the clock constraint ψ , otherwise $\psi(\gamma) = 0$. Given a clock constraint ψ , $\psi = \text{false}$ iff there is no valuation γ such that $\psi(\gamma) = 1$; and $\psi = \text{true}$ iff $\psi(\gamma) = 1$ for every valuation γ . Finally, let $\lambda \subseteq \mathcal{C}$ be a set of clocks, then γ_λ is the clock valuation obtained from γ by setting every clock in λ to 0.

Syntax. Let $\mathcal{L}_3 = \{0, \frac{1}{2}, 1\}$ be the truth set which establishes the specification condition –possible (1), non-possible (0) or unspecified ($\frac{1}{2}$)– of timed behaviors. A MUS-T model $\mathcal{M}$ is a 8-tuple $\langle A, \mathcal{C}, Q, Q_0, E, CS, I_p, I_{np} \rangle$ where:

- A is a finite set of events.
- $\mathcal{C}$ is a finite set of real-valued clocks.
- Q is a finite set of locations which includes the fictitious location q_u , referred to as the unspecified drain.
- $Q_0 \subseteq Q$ is the set of initial locations.
- $E \subseteq Q \times A \times \Psi(\mathcal{C}) \times 2^{\mathcal{C}} \times Q$ is a finite set of edges of the form $(q, \langle a, \psi, \lambda \rangle, q')$ where q, q' are the source and target locations. $\langle a, \psi, \lambda \rangle$ is a timed event which specifies an event $a \in A$, a guard $\psi \in \Psi(\mathcal{C})$ and a subset $\lambda \subseteq \mathcal{C}$ of clocks to be reset upon crossing the edge.
- $CS : E \rightarrow \mathcal{L}_3$ is a total function which assigns a truth value (specification condition) to every edge. An edge $e \in E$ is called possible edge iff $CS(e) = 1$; partially unspecified edge if $CS(e) = \frac{1}{2}$; and non-possible edge iff $CS(e) = 0$.
- I_p and $I_{np} : Q \rightarrow \Psi(\mathcal{C})$ are functions which assign to every location $q \in Q$ a possible invariant $I_p(q)$, establishing when the progress of the time is possible

in the location; and a non-possible invariant $I_{np}(q)$, establishing temporal contexts in which time cannot progress.

Consider $q \in Q$ and $a \in A$, $\Psi_p(q, a) = \{\psi \mid (q, < a, \psi, - >, -) \in E, CS(e) = 1\}$ is called the set of possible guards for the event a in the location q . Similarly, for $CS(e) = 0$, $\Psi_{np}(q, a)$ is called the set of non-possible guards; and for $CS(e) = \frac{1}{2}$, $\Psi_u^P(q, a)$ the set of partially unspecified guards. A MUS-T model is well-formed iff the following conditions hold:

- For every $q \in Q$, $I_p(q) \wedge I_{np}(q) = false$.
- For every $q \in Q$ and for every $a \in A$, (1) $\Psi_p(q, a) \wedge \Psi_{np}(q, a) = false$, (2) $\Psi_p(q, a) \wedge \Psi_u^P(q, a) = false$ and (3) $\Psi_{np}(q, a) \wedge \Psi_u^P(q, a) = false$.

Consider a well-formed MUS-T model $\mathcal{M}$. For every pair $q \in Q$, $a \in A$, the totally-unspecified guard

$$\psi_u^T(q, a) = \neg(\bigvee_{\psi_p \in \Psi_p(q, a)} \psi_p) \wedge \neg(\bigvee_{\psi_{np} \in \Psi_{np}(q, a)} \psi_{np}) \wedge \neg(\bigvee_{\psi_u^P \in \Psi_u^P(q, a)} \psi_u^P)$$

implicitly defines the edge $(q, < a, \psi_u^T, \{\} >, q_u)$. So, for every q , the guards in crossing an a -edge split the valuation space $\mathfrak{R}_+^C$ into three subsets in which the specification condition is possible, non-possible or unspecified. The three subsets are a partition –complete (definition of $\psi_u^T(q, a)$) and disjoint (well-formed rules)–. In the same way, for every q , the unspecified invariant $I_u(q) = \neg I_p(q) \wedge \neg I_{np}(q)$ is defined. The invariants split the valuation space, for the progress of the time. Once again, the three subsets are a partition –complete (by the definition of I_u) and disjoint (well-formed rules)–. The unspecified drain q_u is the fictitious target location of totally unspecified edges in a MUS-T model. q_u is a location that holds $\psi_u^T(q_u, a) = true$, $\forall a \in A$ and $I_u(q_u) = true$. The intuition behind q_u is defining a maximal evolution location. That is, since all the timed behaviors from q_u are unspecified, there are no restrictions in future specifications for this location.

Semantics. The semantics for a MUS-T model $\mathcal{M}$ is given in terms of a dense and many-valued labeled transition system $\mathcal{M}^S = \langle A, \mathcal{C}, \mathcal{S}_M, \mathcal{S}_M^0, \mathcal{T}_M, \mathcal{CS} \rangle$ over the truth set $\mathcal{L}_3$. $\mathcal{S}_M = Q \times \mathfrak{R}_+^C$ is the set of states, i.e., pairs of the form (q, γ) with $q \in Q$ and $\gamma \in \mathfrak{R}_+^C$. $\mathcal{S}_M^0 = \{(q_0, \gamma^0)\}$ is the set of initial states with γ^0 assigning 0 to every clock. $\mathcal{T}_M \subseteq \mathcal{S}_M \times (A \cup \mathfrak{R}_+) \times \mathcal{S}_M$ is the transition relation which identifies a source and a target state $\in \mathcal{S}_M$; and a label $a \in A$ (discrete transition) or $\tau \in \mathfrak{R}_+$ (temporal transition). Finally, $\mathcal{CS} : \mathcal{T}_M \rightarrow \mathcal{L}_3$ is the total function which assigns a specification condition $\mathcal{CS}(t)$ to every transition $t \in \mathcal{T}_M$. For a transition $t \in \mathcal{T}_M \mid c_s = \mathcal{CS}(t)$, we adopt the more intuitive notation $(q, \gamma) \xrightarrow{a, c_s} (q', \gamma')$ or $(q, \gamma) \xrightarrow{\tau, c_s} (q', \gamma')$. $\mathcal{CS}$ is determined by the guards and invariants of the model $\mathcal{M}$. Let (q, γ) be a state, the transition relation is defined by the following rules:

- Time transitions: Given a delay $\tau \in \mathfrak{R}_+$, let γ' be within the range $\gamma \leq \gamma' \leq \gamma + \tau$. A transition $(q, \gamma) \xrightarrow{\tau, c_s} (q, \gamma + \tau)$ exists provided that (1) $I_p(q)(\gamma') = 1$ for every γ' , then $c_s = 1$; or (2) $I_{np}(q)(\gamma') = 1$ for some γ' , then $c_s = 0$; or (3) otherwise $c_s = \frac{1}{2}$.

- Event transitions: Given an edge $e = (q, \langle a, \psi, \lambda \rangle, q') \in E \mid c_s = CS(e)$, a transition $(q, \gamma) \xrightarrow{a, c_s} (q', \gamma_\lambda)$ exists provided that $\psi(\gamma) = 1$.

Definition 1. Deterministic MUS-T. A MUS-T model $\mathcal{M}$ is called deterministic (DMUS-T) iff it has only one start location, $|Q_0| = 1$, and for every pair of edges in E , $(q, \langle a, \psi_1, - \rangle, -)$ and $(q, \langle a, \psi_2, - \rangle, -)$, the clock constraints are mutually exclusive, i.e., $\nexists \gamma \in \mathcal{R}_+^C$ such that $(\psi_1 \wedge \psi_2)(\gamma) = 1$.

3 SCTL-T and SCTL-T^s: Requirements and Scenarios

We will specify the requirements (declarative) and the scenarios (imperative) of a real-time system using the logic SCTL-T (Timed Simple Causal Temporal Logic). SCTL-T is a dense-time extension of the causal and many-valued temporal logic SCTL [9]. SCTL-T formulas match the causal pattern **Premise** $\otimes$ **Consequence** which establishes a causing condition (Premise); a temporal operator which determines the applicability of the cause ($\otimes$); a condition which is the effect (Consequence); and a quantifier ($\mathcal{A}$) which determines the degree of satisfaction of the consequence in the applicability set. After introducing the quasi-boolean algebra $\mathcal{L}_6$ (section 3.1), the syntax and the semantics of the logic are defined both declaratively (section 3.2) and imperatively (section 3.4)².

3.1 MPU: Algebra of Middle Point Uncertainty

SCTL-T semantics is given over the partially ordered set of truth values $(\mathcal{L}_6, \leq)$ where $\mathcal{L}_6 = \{0, \frac{1}{4}, \frac{1}{2}, \frac{3}{4}, 1\}$ (Hasse diagram of figure 1). Every two elements $a, b \in \mathcal{L}_6$ have a least upper bound $a \vee b = \sup\{a, b\}$, and a greatest lower bound $a \wedge b = \inf\{a, b\}$. Besides, the unary operation $\neg$ is defined by horizontal symmetry. The 4-tuple $L_6 = (\mathcal{L}_6, \wedge, \vee, \neg)$ has the structure of a quasi-boolean algebra called algebra of **M**iddle **P**oint **U**ncertainty (MPU).

Truth values $\in \mathcal{L}_6$ have its origins in the MUS-T specification condition $\in \mathcal{L}_3$ and the causal operation $\rightarrow$ over $\mathcal{L}_6$ (see figure 1) defined below:

$$a, b \in \mathcal{L}_6, \rightarrow (a, b) = (a \vee \frac{1}{2}) \wedge \left(\left(\neg a \wedge \frac{1}{2} \right) \vee b \right)$$

that is, $\rightarrow$ is interpreted as the usual implication when $a > \frac{1}{2}$. If $a \leq \frac{1}{2}$, the result is $\frac{1}{2}$ (non-applicable) which, intuitively, stands for causal propositions where the premise cannot be satisfied. The intuitive meaning of truth values (table of figure 1) is made clear in section 3.3.

² For shortness the reader is referred to [6] where the fixpoint characterization of SCTL-T is included. For a practical specification without explicit fixpoint constructions we provide a set of such constructions as predefined macros: $\mathcal{G}$, *it is always Going to be the case*; $\mathcal{F}$, *at least once in the Future*; and so on.

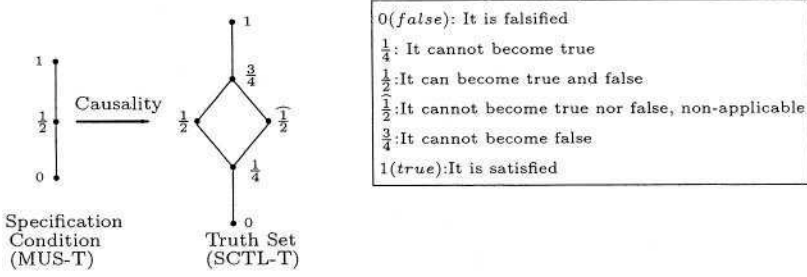


Fig. 1. SCTL-T Truth values.

3.2 SCTL-T: Expressing Requirements

Syntax. A SCTL-T declaration $I_R := \phi$ establishes an identifier I_R , which ranges over a finite set of requirement identifiers $\mathcal{I}_R$, and a SCTL-T formula $\phi \in \Phi_{SCTL-T}$. Φ_{SCTL-T} is given by the following grammar:

$$\begin{aligned} \langle \Phi_{SCTL-T} \rangle &::= \Phi \forall \otimes \Phi \mid \Phi \exists \otimes \Phi \\ \langle \Phi \rangle &::= \psi \mid a \mid \xi \mid y. \Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg \Phi \mid \Phi_{SCTL-T} \end{aligned}$$

where $\mathbb{E} = \{\forall, \exists\}$ are the usual path quantifiers adapted to many-valued reasoning; and $\otimes$ is a temporal operator in the set $\Theta = \{\Rightarrow, \Rightarrow_+, \Rightarrow_-, \Rightarrow \bigcirc, \Rightarrow \odot\}$. Clock variables are model clocks $\mathcal{C}$ of a MUS-T model and specification clocks $\mathcal{E}$ which are defined in the formula. A specification clock $y \in \mathcal{E}$ is a clock variable which is bound and reset in a formula ϕ by $y. \phi$, the *reset quantifier*. SCTL-T grammar contains three kinds of atomic propositions: $a \in A$ is an event proposition in the alphabet A of a MUS-T model; $\xi \in \Xi$ is a truth proposition in the set $\Xi = \{true, false, \emptyset\}$; and $\psi \in \Psi(\mathcal{C} \cup \mathcal{E})$ is a time proposition over model clocks $\mathcal{C}$ and specification clocks $\mathcal{E}$.

Applicability set and Quantified causality. A temporal operator $\otimes \in \Theta$ fixes the relative order between the state (q, γ) in which the premise is formulated, and the states in the scope of the consequence (applicability set, $\perp (\otimes, (q, \gamma))$). $\mathbb{E}$ quantification determines the degree of satisfaction which is required in the applicability set. In quantified causality ($\mathbb{E} \otimes$), the truth degree of the quantified consequence depends on the specification conditions of the transitions which make the applicability states accessible (accessibility condition, c_s) in the causal formula (see the following section for an intuitive description). The applicability set is defined as follows:

$$\begin{aligned} \perp (\Rightarrow \bigcirc, (q, \gamma)) &= \{(q', \gamma', c_s) \mid (q, \gamma) \xrightarrow{a, c_s} (q', \gamma')\} \\ \perp (\Rightarrow \odot, (q, \gamma)) &= \{(q', \gamma', c_s) \mid (q', \gamma') \xrightarrow{a, c_s} (q, \gamma)\} \\ \perp (\Rightarrow_+, (q, \gamma)) &= \{(q', \gamma', c_s) \mid (q, \gamma) \xrightarrow{\tau, c_s} (q', \gamma')\} \\ \perp (\Rightarrow_-, (q, \gamma)) &= \{(q', \gamma', c_s) \mid (q', \gamma') \xrightarrow{\tau, c_s} (q, \gamma)\} \\ \perp (\Rightarrow, (q, \gamma)) &= \{(q, \gamma, 1)\} \end{aligned}$$

Consider a SCTL-T declaration $I_R := \phi$, I_R is called a (non-strict) future formula provided that every temporal operator mentioned in the formula ϕ is in $\{\Rightarrow_+, \Rightarrow \bigcirc\}$ ($\{\Rightarrow_+, \Rightarrow \bigcirc, \Rightarrow\}$). Similarly, I_R is called a (non-strict) past formula provided that every temporal operator mentioned in the formula ϕ is in $\{\Rightarrow_-, \Rightarrow \odot\}$ ($\{\Rightarrow_-, \Rightarrow \odot, \Rightarrow\}$).

Semantics. Given a MUS-T model $\mathcal{M}$, a SCTL-T formula is interpreted with respect to the semantic model $\mathcal{M}_{\mathcal{E}}^S$ induced by $\mathcal{M}$ where $\mathcal{C}$ is extended by all clocks $\mathcal{E}$ mentioned in the formula. The satisfaction relation

$$\models : (\Phi_{SCTL-T} \times (Q \times \mathbb{R}_+^{C \cup \mathcal{E}} \times \mathcal{L}_3)) \longrightarrow \mathcal{L}_6$$

assigns a truth value $\in \mathcal{L}_6$, $\models (\phi, (q, \gamma, c_s))$, to a formula $\phi \in \Phi_{SCTL-T}$ evaluated in a state (q, γ) with accessibility condition c_s . $\models$ is inductively defined as follows:

- i) $\models (\psi, (q, \gamma, c_s)) \equiv \psi(\gamma)$
- ii) $\models (a, (q, \gamma, c_s))$ is 1 iff $(q, \gamma) \xrightarrow{a, 1}$, $\frac{1}{2}$ iff $(q, \gamma) \xrightarrow{a, \frac{1}{2}}$, and 0 iff $(q, \gamma) \xrightarrow{a, 0}$
- iii) $\models (true, (q, \gamma, c_s)) = c_s$, $\models (false, (q, \gamma, c_s)) = \neg c_s$, $\models (\emptyset, (q, \gamma, c_s)) = 1$
- iv) $\models (y.\phi, (q, \gamma, c_s)) \equiv \models (\phi, (q, \gamma_{\{y\}}, c_s))$
- v) $\models (\phi \wedge \phi', (q, \gamma, c_s)) \equiv \models (\phi, (q, \gamma, c_s)) \wedge \models (\phi', (q, \gamma, c_s))$
- vi) $\models (\phi \vee \phi', (q, \gamma, c_s)) \equiv \models (\phi, (q, \gamma, c_s)) \vee \models (\phi', (q, \gamma, c_s))$
- vii) $\models (\neg\phi, (q, \gamma, c_s)) \equiv \neg(\models (\phi, (q, \gamma, c_s)))$
- viii) $\models (\phi \otimes \phi', (q, \gamma, c_s)) \equiv$
 $\rightarrow \left(\models (\phi, (q, \gamma, c_s)), \bigwedge_i (\neg c_{s_i}' \vee \models (\phi', (q_i', \gamma_i', c_{s_i}')) \right)$
 where $(q_i', \gamma_i', c_{s_i}') \in \perp (\otimes, (q, \gamma))$
- ix) $\models (\phi \exists \otimes \phi', (q, \gamma, c_s)) \equiv$
 $\rightarrow \left(\models (\phi, (q, \gamma, c_s)), \bigvee_i (c_{s_i}' \wedge \models (\phi', (q_i', \gamma_i', c_{s_i}')) \right)$
 where $(q_i', \gamma_i', c_{s_i}') \in \perp (\otimes, (q, \gamma))$

3.3 The Intuition behind SCTL-T Truth Values

In this section we provide an intuitive description of the SCTL-T truth values. When the available information is incomplete, like in a MUS-T model, computing the degree of satisfaction of a requirement in a precise way (true or false) is generally impossible, and intermediate truth values accounts for the imperfect knowledge of truth. These intermediate truth values are really degrees of uncertainty and express propensity to be true and (or) propensity to be false. Informally, truth values in SCTL-T express propensity or capability of a MUS-T model to satisfy a causal requirement in future iterations of the SCTL/MUS-T process.

Atomic propositions in a SCTL-T formula are events in the alphabet A of a MUS-T model $\mathcal{M}$ under study, or time propositions, which we do not consider in this informal introduction. In a model $\mathcal{M}$, the information about a proposition a can be known with certainty (as true – 1 –, or as false – 0 –) or unknown (unspecified – $\frac{1}{2}$ –). We now show how the incomplete information in $\mathcal{M}$ results

in uncertainty (propensity to be true or propensity to be false) when evaluating the truth. Let a, b be events $\in A$ and $a \wedge b$ be a logic formula which is evaluated in a state s of $\mathcal{M}$. Consider the following informal examples:

- a and b are possible in s , the information about a and b is complete and the truth of the formula is known with certainty (true, truth value 1).
- a is non-possible in s , the truth of the formula is known with certainty (false, truth value 0) despite the fact the information about b would be incomplete.
- a is possible and b is unspecified in s , the truth of the formula is totally unknown. The formula is both a potential fulfillment (true) and a potential unfulfillment (false). So, the truth of the formula expresses really the uncertainty (truth value $\frac{1}{2}$) about its truth or falseness.

However intermediate values in this 3-valued logic arise if we account the notion of causality in SCTL-T. The causal intuition says that most facts about the world (intended system) tend to stay the same over time, unless an action takes place which affects them. Consider the informal logic formula “ a causes b ” which is evaluated in a state s . This formula reflects the causal intuition that “If a is possible in s , b is also possible in s . If not, a does not affect b ”. The most simple cases arise when the information in the model $\mathcal{M}$ about a and b is enough to establish the truth with certainty:

- a is possible and b is possible, the truth value is 1 (causal fulfillment).
- a is possible and b is non-possible, the truth value is 0 (causal unfulfillment).
- a is non-possible, meanwhile in a logic with implication the truth value would be 1, with a causal interpretation, the formula does not make sense in s since the event which affects b does not take place. That is, the formula is non-applicable, truth value $\frac{1}{2}$ (nonsense fulfillment).

On the contrary, if the information about a and b in $\mathcal{M}$ is not enough to establish the truth with certainty, additional uncertainty values arise:

- b is non-possible and a is unspecified. In a future iteration in the SCTL/MUS-T process, the information about a in $\mathcal{M}$ will be provided. a could be specified as possible or as non-possible and so, the causal formula could be an unfulfillment or a nonsense fulfillment. Despite the fact the information in $\mathcal{M}$ is incomplete, the formula is known to be impossible to satisfy, being a potential unfulfillment (truth value $\frac{1}{4}$). $\frac{1}{4}$ accounts for a lower level of uncertainty than $\frac{1}{2}$ and expresses propensity to be false.
- b is possible and a is unspecified. Again, in a future iteration, a could be specified as possible or as non-possible and so, the formula could be a fulfillment or a nonsense fulfillment. That is, the formula is known to be impossible to falsify, being a potential fulfillment (truth value $\frac{3}{4}$).

Quantified causality accounts for the same notion of truth as follows. Consider the figures 2 being fragments of an abstract state space of a MUS-T model $\mathcal{M}$. Consider a generic existential requirement $R_{\exists} := \phi \exists \otimes \phi'$, evaluated in an abstract state s (marked with $\circ$), and $\{s_a\}$ the applicability set (marked with $\bullet$). The most simple cases arise when the information in the model $\mathcal{M}$ (even if

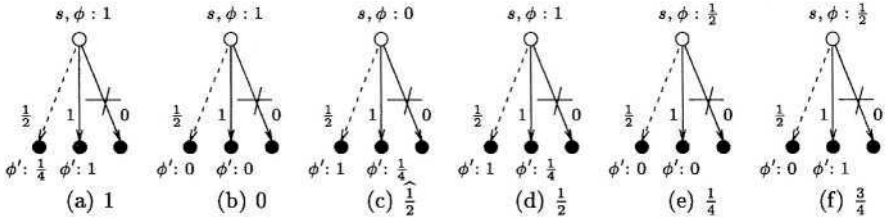


Fig. 2. Existentially-quantified requirement ($R_{\exists} := \phi \exists \otimes \phi'$): degrees of satisfaction

incomplete) is enough to establish the truth with certainty. Such is the case in the examples 2(a), 2(b), and 2(c) where the knowledge level is maximal. That is, whatever incremental specification from this abstract model, the truth value of $R_{\exists}$ will remain the same. Remark that, turning the unspecified transition (in these examples) into possible or non-possible does not alter the truth of $R_{\exists}$.

If the information in $\mathcal{M}$ is not enough to establish the truth with certainty, additional truth values (uncertainty values actually) arise. In the example 2(d), the truth value of $R_{\exists}$ is totally unknown ($\frac{1}{2}$). $R_{\exists}$ can become (1) a satisfied requirement, if the unspecified transition turns into a possible one; (2) a falsified requirement, if the unspecified transition turns into a non-possible one and ϕ' evolves to 0; and (3) a non-applicable requirement, if ϕ' evolves to $\frac{1}{2}$. Examples for intermediate levels of certainty or knowledge are the following:

- (e) $R_{\exists}$ is impossible to satisfy ($\frac{1}{4}$) since the quantified consequence ϕ' is falsified. In a future iteration of the process, $R_{\exists}$ can evolve to a non-applicable requirement, if ϕ is falsified; or to a falsification, if ϕ is satisfied. Despite the fact the information in $\mathcal{M}$ is incomplete, $R_{\exists}$ is known to be impossible to satisfy, being a potential unfulfillment (propensity to be false).
- (f) $R_{\exists}$ is impossible to falsify ($\frac{3}{4}$) since the quantified consequence ϕ' is satisfied. Again, in a future iteration, $R_{\exists}$ can evolve to a non-applicable requirement, if ϕ is falsified; or to a satisfaction, if ϕ is satisfied. $R_{\exists}$ is known to be impossible to falsify, being a potential fulfillment (propensity to be true).

Finally, consider a generic universal requirement $R_{\forall} := \phi \forall \otimes \phi'$ and see figure 3 for examples of levels of knowledge.

3.4 SCTL-T^s: Expressing Scenarios

Using scenarios a designer proposes a situation and decides what behavior would be appropriate to that context. We formalize scenarios defining a synthesis-oriented version of SCTL-T (SCTL-T^s) with imperative future semantics (METATEM [10]). The general idea behind imperative future is to rewrite each scenario to be synthesized into a set of SCTL-T^s formulas of the form:

declarative (past formula) causes imperative (future formula)

and then treat such formulas as synthesis rules showing how the future (imperative) can be constructed given the past constructed so far (declarative).

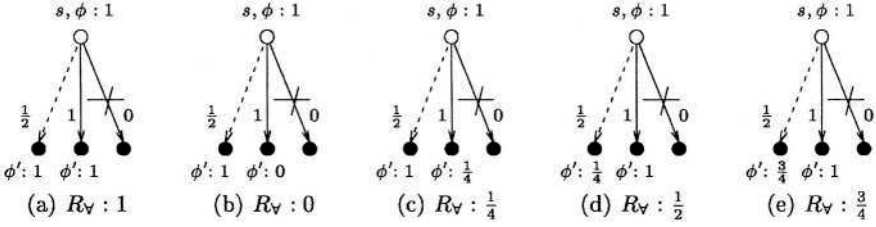


Fig. 3. Universally-quantified requirement ($R_{\forall} := \phi \ \forall \otimes \phi'$): degrees of satisfaction

Syntax. A synthesis rule $I_S := \phi$ establishes (1) an identifier I_S , which ranges over a finite set of scenario identifiers $\mathcal{I}_S$; and (2) a causal formula $\phi \in \Phi_{SCTL-T^*}$, where the premise specifies the synthesis context and the consequence specifies the new non-strict future behavior. Φ_{SCTL-T^*} is given by the following grammar:

$$\begin{aligned} \langle \Phi_{SCTL-T^s} \rangle &::= ini \ \forall \otimes_f \Phi_I \mid \Phi_p \forall \otimes_f \Phi_I \\ \langle \Phi_I \rangle &::= \psi \mid a \mid \neg a \mid \xi \mid y. \Phi_I \mid a\{x\} \mid \Phi_I \wedge \Phi_I \mid \Phi_I \forall \otimes_f \Phi_I \end{aligned}$$

where Φ_p is the non-strict past Φ_{SCTL-T} ; *ini*, a special symbol which stands for the initial state of a deterministic model; and $\otimes_f$, a non-strict future temporal operator in Θ . For what concerns the imperative semantics of Φ_I , we restrict our attention to deterministic MUS-T models (DMUS-T, definition 1) and DSF (*Deterministic Safetyformulas*) [11]. Restricting for synthesis is based on the fact that in the case of synthesizing a liveness property, any finite state sequence may be extended to an infinite one that satisfies it. Consequently, liveness properties have to be implemented by explicit strategies expressed as SCTL-T^s scenarios. Remark that, Φ_I grammar introduces an imperative form of bind $a\{x\}$ to allow the explicit management of model clocks from the logic. Given a model, $a\{x\}$ enforces: (1) adding a clock x , if it does not exist, to the model clocks $\mathcal{C}$ and (2) resetting the clock in an a -labeled discrete transition.

Imperative Semantics. The imperative semantics for Φ_{SCTL-T^*} proceeds selecting synthesis contexts (specified by the declarative part) and then adding, in these contexts, the new future behavior specified by the imperative part (enforcing a truth value 1). An incremental process entails restricting imperative semantics to specify transitions which are unspecified (**successful synthesis**), that is, a specification condition $\frac{1}{2}$ turns into 1 or 0. Any other change in the specification condition reflects an **inconsistency failure** which should identify the scenarios in conflict.

Definition 2. Applicable rule and synthesis context. Let I_S be a synthesis rule in conjunctive form, $I_S := \phi_p \otimes_f \phi_I^1 \wedge \dots \wedge \phi_I^n$, which is interpreted over a DMUS-T model $\mathcal{M}$. If the declarative part $\phi_p = \text{ini}$, I_S is an applicable rule in the only synthesis context (q_0, γ^0) . Otherwise I_S is an applicable rule in

$(q, \gamma) \in \mathcal{S}_{\mathcal{M}}$ iff the declarative part holds $\models (\phi_p, (q, \gamma, 1)) = 1$, and (q, γ) is said to be a synthesis context for the rule I_S .

Definition 3. Feasible Synthesis. Let (q, γ) be a state $\in \mathcal{S}_{\mathcal{M}}$ of a DMUS-T model $\mathcal{M}$ with accessibility condition c_s referred to as $s = (q, \gamma, c_s)$. An element ϕ_I^i of the imperative conjunction $\phi_I^1 \wedge \dots \wedge \phi_I^n$ is feasible synthesis in s iff $\models (\phi_I^i, s) \geq \frac{1}{2}$, that is, it is feasible to update $\mathcal{M}$ such that $\models (\phi_I^i, s) = 1$.

Definition 4. Successful Synthesis and Inconsistency Failure. Let $(q, \gamma) \in \mathcal{S}_{\mathcal{M}}$ be a state of a DMUS-T model $\mathcal{M}$, and $I_S := \phi_p \otimes_f \phi_I^1 \wedge \dots \wedge \phi_I^n$ be a synthesis rule. I_S is said to be successful iff for every applicability state $s_a \in \perp (\otimes_f, (q, \gamma))$, every ϕ_I^i is feasible synthesis in s_a ($\models (\phi_I^i, s_a) \geq \frac{1}{2}$) or non-applicable in s_a ($\models (\phi_I^i, s_a) = \frac{1}{2}$). Otherwise, I_S is an inconsistency failure.

Imperative semantics of atomic elements. Let $s = (q, \gamma, c_s)$ be a state of a DMUS-T model $\mathcal{M}$ with accessibility condition c_s , then:

- a : Let $t \in \mathcal{T}_{\mathcal{M}}$ be the transition leaving (q, γ) and labeled by a , a is feasible synthesis iff $\mathcal{CS}(t) = 1$ or $\mathcal{CS}(t) = \frac{1}{2}$. Otherwise, $\mathcal{CS}(t) = 0$, a is not feasible synthesis. In case $\mathcal{CS}(t) = \frac{1}{2}$, the specification condition of the transition is turned into 1, that is, $\mathcal{CS}(t) = \frac{1}{2} \mapsto 1$.
- $\neg a$: The reverse imperative semantics.
- $\emptyset$: $\emptyset$ is always feasible synthesis since $\models (\emptyset, (q, \gamma, c_s)) = 1$.
- $true$: $true$ is feasible synthesis iff $c_s = 1$ or $c_s = \frac{1}{2}$. Otherwise, $c_s = 0$, $true$ is not feasible synthesis. In case $c_s = \frac{1}{2}$, let $t \in \mathcal{T}_{\mathcal{M}}$ be the transition which makes (q, γ, c_s) accessible, then $\mathcal{CS}(t) = \frac{1}{2} \mapsto 1$.
- $false$: The reverse imperative semantics.
- ψ : ψ is feasible synthesis iff $\models (\psi, (q, \gamma, c_s)) = 1$. Otherwise, ψ is not feasible.
- $y.\phi_I$: $y.\phi_I$ is feasible synthesis iff ϕ_I is feasible synthesis in $(q, \gamma_{\{y\}}, c_s)$.
- $a\{x\}$: See the imperative semantics below.

Imperative Semantics of $a\{x\}$ elements. Informally an element $a\{x\}$ is feasible synthesis if the clock x is a new clock in the DMUS-T model $\mathcal{M}$ or if $x \in \mathcal{C}$ but resetting the clock does not alter the specified behaviors in the model.

Definition 5. Active clocks. Let $I(q) = \{I_p(q), I_{np}(q)\}$ be the invariants which are not unspecified in a location q , and let $\Psi_A(q) = \{\Psi_p(q, a), \Psi_{np}(q, a), \Psi_u^P(q, a) \mid a \in A\}$ be the guards which are not totally-unspecified in a location q . For a DMUS-T model $\mathcal{M}$ and a location $q \in \mathcal{Q}$, $\text{Act}(q) := \text{Act}L(q) \bigcup_{q' \in \text{suc}(q)} \text{Act}(q')$ is called the set of active clocks in q ,

where $\text{Act}L(q) = \{x \mid x \in I(q) \text{ or } x \in \Psi_A(q)\}$ and $\text{suc}(q)$ is the set of possible or partially-unspecified successors in E .

Let $s = (q, \gamma, c_s)$ be a state of a DMUS-T model $\mathcal{M}$ with accessibility condition c_s , $a\{x\}$ is feasible synthesis iff a is feasible synthesis and $x \notin \text{Act}(q')$, where q' is the location such that $(q, \gamma) \xrightarrow{a, c_s'} (q', \gamma') \in \mathcal{T}_{\mathcal{M}}$. If $a\{x\}$ is feasible synthesis, then (1) the imperative semantics of a is added and (2) x is reset upon crossing the a -labeled transition.

Imperative Semantics of causal elements. Let $s = (q, \gamma, c_s)$ be a state of a DMUS-T model $\mathcal{M}$ with accessibility condition c_s , and consider a causal element $\phi_I \forall \otimes_f \phi_I'$. If $\models (\phi_I, (q, \gamma, c_s)) \leq \frac{1}{2}$, the causal element is non-applicable. Otherwise, $\models (\phi_I, (q, \gamma, c_s)) \geq \frac{1}{2}$, consider the truth value of the causal element $\rightarrow \left(\models (\phi_I, (q, \gamma, c_s)), \bigwedge_i \neg c_{s_i}' \vee \models (\phi_I', (q_i', \gamma_i', c_{s_i}')) \right)$. The premise (feasible synthesis) is synthesized by enforcing $\models (\phi_I, (q, \gamma, c_s)) = 1$. The causal element is feasible synthesis iff for every $(q_i', \gamma_i', c_{s_i}')$ one of the following holds:

- $c_{s_i}' = 0$. No imperative semantics is added.
- $c_{s_i}' = 1$ or $c_{s_i}' = \frac{1}{2}$ and the consequence ϕ_I' is feasible synthesis in $(q_i', \gamma_i', c_{s_i}')$; the consequence ϕ_I' is synthesized (if feasible) by enforcing $\models (\phi_I', (q_i', \gamma_i', c_{s_i}')) = 1^3$. Otherwise the causal element is not feasible.

4 The Rationale Behind the Many-Valued Approach

SCTL/MUS-T is iteratively and incrementally articulated as subsequent iterations consisting of (figure 4(b)): formalizing the specification which captures the system requirements (as SCTL-T declarative requirements), incremental design of the system by means of formalizing typical behaviors as scenarios (as SCTL-T^s imperative synthesis rules); and verification and validation of design decisions conforming to the requirements specification. The system is iteratively designed and a model-oriented specification, the MUS-T model, is obtained by incrementally incorporating SCTL-T^s scenarios (**incremental synthesis**). Besides, SCTL-T requirements are **model checked** in the current incomplete MUS-T model, the one in the current iteration, in order to decide: if the system already satisfies the new requirements; if it is not able to provide, in a future iteration, these requirements from the current design (**inconsistency**); or, if the system does not satisfy the requirements, but it is able to do it in a future iteration (**incompleteness**).

Figure 5 shows an abstract instance of a SCTL/MUS-T specification process, that is, an instance of figure 4(b). The traditional declarative view is covered by SCTL-T requirements and the imperative view is provided by SCTL-T^s scenarios. In this way, the methodology provides both a traditional declarative view (“what”) and a constructive imperative view (“how”). An instance of the SCTL/MUS-T lifecycle is a sequence of iterations where a very-abstract prototype of the intended system is designed scenario by scenario. As we argued above, uncertainty and disagreement are pervasive and desirable in this process. On the one hand, uncertainty is a constant feature throughout the instance, since the model-oriented specification of the system is incomplete (MUS-T model). Such incompleteness results in truth values with more or less level of uncertainty. So, given a requirement, not only fulfillment or unfulfillment, but potential fulfillments, potential unfulfillments and even impossible-to-decide results can be

³ If $c_s = \frac{1}{2}$, $c_{s_i}' = \frac{1}{2} \mapsto 0$ is feasible, but this imperative semantics entails trivial solutions, without successors. Besides $c_{s_i}' = \frac{1}{2} \mapsto 1$ is feasible, but this imperative semantics entails maximal models in which every successor is possible.

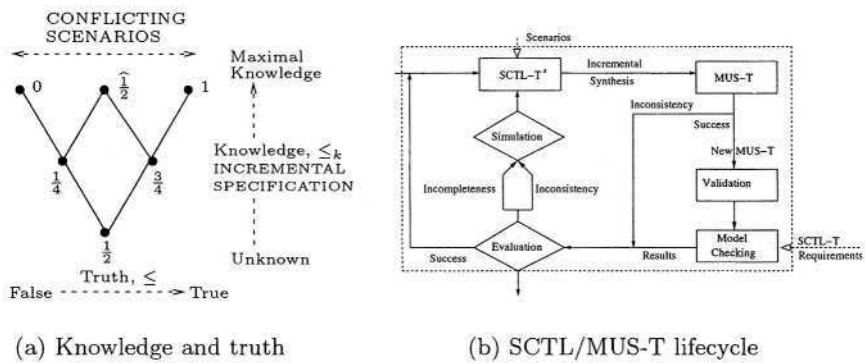


Fig. 4. Incremental gain for incremental effort

obtained. On the other hand, disagreement can appear between one iteration and the next, as a form of consistency checking, when a new scenario is incorporated which conflicts with a previous scenario or scenarios.

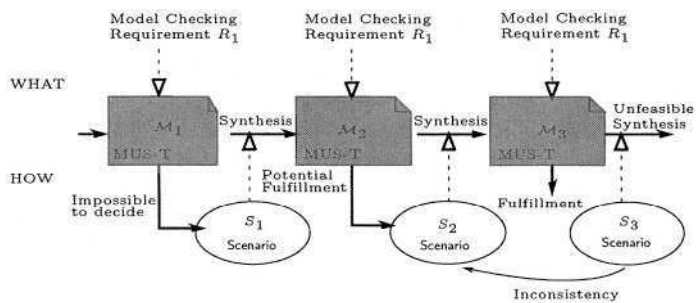


Fig. 5. Uncertainty and Disagreement

Identifying a new scenario the system should exhibit implicitly accounts for a better understanding about the system which is being specified. A common sense thought can be the following. The more scenarios are identified, the more knowledge about the system is gained. So, the model checking problem established as “does the system satisfy a SCTL-T requirement?” should be answered with a greater level of certainty or, at least, not with a lower level. Regarding this point, the many-valued approach allows us to formalize the knowledge about the properties of the system being specified. That is, more is known about the properties as more is iteratively and incrementally specified about the system.

Consider the double Hasse diagram of figure 4(a), there are two partial orderings displayed there: the $\leq$ ordering is, intuitively, one on degree of truth; the $\leq_k$ ordering is on degree of information or knowledge. As it is shown in section 5, the level of knowledge of a generic SCTL-T requirement never decreases when a new scenario SCTL-T^s is synthesized (see “incremental specification” in figure

4(a)). The above fact brings some information about the evolution of the truth value of the requirements in the process. There are three values, $\{0, \frac{1}{2}, 1\}$, that are maximal in the ordering $\leq_k$, that is, its knowledge level is maximal and the degree of satisfaction will remain equal in future iterations of the lifecycle. There is one, $\frac{1}{2}$, that is the smallest, the degree of satisfaction in future iterations of the lifecycle is totally unknown. Finally, there are two intermediate values, $\frac{1}{4}$ and $\frac{3}{4}$, that is, its knowledge level is intermediate and the degree of satisfaction in future iterations is somewhat restricted: $\frac{3}{4}$ means that the degree of satisfaction only will be $\frac{3}{4}$, $\frac{1}{2}$ or 1, that is, the formula is impossible to falsify (incompleteness failure); on the contrary, $\frac{1}{4}$ makes the formula impossible to satisfy, the degree of satisfaction only can remain $\frac{1}{4}$ or evolve to $\frac{1}{2}$ or 0 (inconsistency failure).

One central point is the effects of an unsuccessful synthesis of a SCTL- T^s scenario. When it is the case, the knowledge about the system does not actually increase, but an erroneous knowledge is replaced by a more confident one. So, the process stops being incremental and some requirements (the ones affected by the misunderstanding) can break the rules of evolution above (see “conflicting scenarios” in figure 4(a)).

The dynamics of the process is guided by the many-valued results provided by the automated tasks in the process: model checking and incremental synthesis. In order to receive feedback from the process, instead of just pointing out inconsistency and incompleteness failures, additional information is provided to assist in resolving them:

- A SCTL- T scenario is successfully synthesized provided that the truth value of its imperative part is $\geq \frac{1}{2}$. Otherwise the imperative part is $\frac{1}{2}$ (non-applicable) or $< \frac{1}{2}$ (inconsistency failure). In the last case the scenarios in conflict are supplied; this information allows the designer to inspect what scenarios are error-prone and to uncover a misunderstanding of the system.
- If a SCTL- T requirement is not satisfied (< 1) the designer is guided as follows. In case of inconsistency failure ($< \frac{1}{2}$) counterexamples are computed; by simulating the counterexamples the designer inspects which scenarios, or maybe the requirement, are error-prone. In case of incompleteness failure ($\geq \frac{1}{2}$) completion suggestions are computed, then the model, extended with the supplied suggestions, can be animated. The animation allows the designer to explore what alternative conforms to wishes and to discover new scenarios.

5 Formalizing the Iterative and the Incremental Nature

In this section we provide the formalization of the iterative and incremental nature in a SCTL/MUS- T process.

Iterative view: relation among incomplete models. The iterative nature of the specification process results in a set of incomplete DMUS- T models obtained by synthesizing scenarios in subsequent iterations of the lifecycle. In this section we search for the relation among these DMUS- T models. Reducing the amount

of unspecification by incremental synthesis produces more detailed or specified DMUS-T models. Informally, a more specified DMUS-T model should contain all the specified behaviors (possible and non-possible) in the less specified DMUS-T model and additional specified behaviors.

Definition 6. Let $\mathcal{M}$ and $\mathcal{M}'$ be DMUS-T models, a binary relation $\sqsubseteq_E$ on the states of $\mathcal{M}$ and $\mathcal{M}'$, $\sqsubseteq_E \subseteq \mathcal{S}_{\mathcal{M}} \times \mathcal{S}_{\mathcal{M}'}$, is a *spEcified Time Abstracting Simulation*, ETaS, iff for all states $s \sqsubseteq_E s'$, where $s = (q, \gamma)$, $s' = (q', \gamma')$, and all labels $\tau \in \mathbb{R}_+$ are replaced by $\epsilon \notin (A \cup \mathbb{R}_+)$, the following conditions hold:

$$\begin{aligned} \forall s_1 \mid s \xrightarrow{a, c_s} s_1 \in \mathcal{T}_{\mathcal{M}}, \exists s_1' \mid s' \xrightarrow{a, c_s'} s_1', c_s \leq_k c_s' \text{ and } s_1 \sqsubseteq_E s_1'; \\ \forall s_1 \mid s \xrightarrow{\epsilon, c_s} s_1 \in \mathcal{T}_{\mathcal{M}}, \exists s_1' \mid s' \xrightarrow{\epsilon, c_s'} s_1', c_s \leq_k c_s' \text{ and } s_1 \sqsubseteq_E s_1' \end{aligned}$$

The ETaS is the basis to establish a specification ordering which reflects the iterative nature of the specification process: ETaS relates the DMUS-T models which are obtained by incremental synthesis in the SCTL/MUS-T methodology.

Definition 7. Let $\mathcal{M}$ and $\mathcal{M}'$ be DMUS-T models, $\mathcal{M}'$ *simulates under specification* $\mathcal{M}$, $\mathcal{M} \sqsubseteq_E \mathcal{M}'$, iff it exists an ETaS in the states of $\mathcal{M}$ and $\mathcal{M}'$, $\sqsubseteq_E \subseteq \mathcal{S}_{\mathcal{M}} \times \mathcal{S}_{\mathcal{M}'}$, such that $(q_0, \gamma^0) \sqsubseteq_E (q_0', \gamma^0')$; and $\sqsubseteq_E$ preserves the set the clock constraints $\Psi_{\mathcal{M}}$. $\Psi_{\mathcal{M}}$ is the subset of $\Psi(\mathcal{C})$ obtained by restricting the integer constants to the maximal constants in the model $\mathcal{M}$.

Lemma 1. Let $\mathcal{M}$ be a DMUS-T model, and I_S a scenario, the model $\mathcal{M}'$ obtained by the incremental synthesis of I_S holds $\mathcal{M} \sqsubseteq_E \mathcal{M}'$.

Proof. Hint on proof. Consider $\mathcal{M}$ and $\mathcal{M}' = \mathcal{M}$, then $\mathcal{M} \sqsubseteq_E \mathcal{M}'$. The proof is based on the fact that the imperative semantics for SCTL-T^s (changes in $\mathcal{M}'$) does not alter the above relation.

Incremental view: growing the knowledge. The synthesis scenario by scenario means the loss of some unspecification in the model $\mathcal{M}$. This kind of incremental evolution guarantees the knowledge about the system never decreases.

Lemma 2. Let $\mathcal{M}$ and $\mathcal{M}'$ be DMUS-T models such that $\mathcal{M} \sqsubseteq_E \mathcal{M}'$ and $I_R := \phi$ be a SCTL-T requirement, then the knowledge level of I_R does not decrease in $\mathcal{M}'$. That is, let (q, γ) and (q', γ') be a pair of states $\in \mathcal{S}_{\mathcal{M}}$ such that $(q, \gamma) \sqsubseteq_E (q', \gamma')$, then $\models (\phi, (q, \gamma, c_s)) \leq_k \models (\phi, (q', \gamma', c_s))$ for every $c_s \in \mathcal{L}_3$.

Proof. Hint on proof. If $(q, \gamma) \sqsubseteq_E (q', \gamma')$, then the knowledge level of the degree of satisfaction of atomic elements, $\phi_a \in A \cup \Psi(\mathcal{C} \cup \mathcal{E}) \cup \Xi$, never decreases, that is, $\models (\phi_a, (q, \gamma), c_s) \leq_k \models (\phi_a, (q', \gamma'), c_s)$. Since MPU operators are $\leq_k$ monotonic, by structural induction, $\models (\phi, (q, \gamma, c_s)) \leq_k \models (\phi, (q', \gamma', c_s))$ for every $\phi \in \Phi_{SCTL-T}$, $c_s \in \mathcal{L}_3$.

Note that the above result guarantees the maintenance of the requirements from one iteration to another. A requirement I_R , which is true (false) in a model $\mathcal{M}$, also will be true (false) in the more specified $\mathcal{M}'$ obtained by incorporating a new scenario I_S . That is because true (1) and false (0) are maximal in $\leq_k$.

6 Some Abstract Examples

SCTL/MUS-T is a dense-time methodology, so model checking and incremental synthesis involves obtaining an exact and finite abstraction of the state space (see section 7). In this section, we introduce some abstract examples in order to make clear the imperative semantics of SCTL-T^s. See [7] for a complete case study.

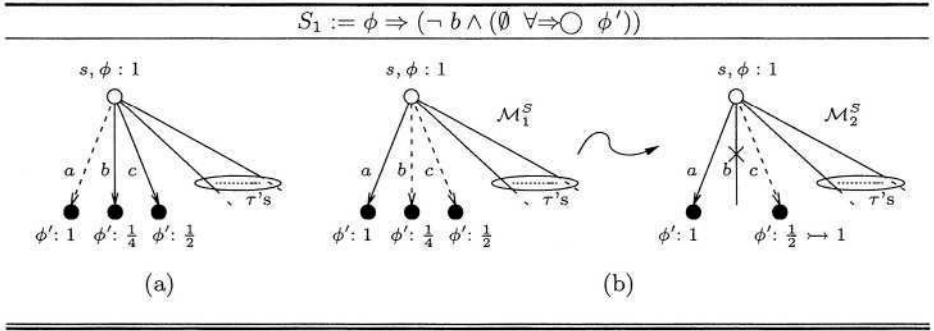


Fig. 6. Imperative Semantics (updating the guards in a DMUS-T model): S_1 is an inconsistency failure in (a) and successful synthesis in (b) where the actions performed in the abstract model are shown

Consider the graphs in figures 6 and 7 being fragments of the state space of an incomplete model (dashed lines are unspecified transitions, solid lines are possible transitions, and crossed lines are non-possible transitions). In the figure 6, the synthesis rule S_1 is applied to the state $s = (q, \gamma, c_s)$, a synthesis context for S_1 . In the case of the figure 6(a), S_1 is an inconsistency failure since the element $\neg b$, in the imperative consequence, is not feasible synthesis. In the case of the figure 6(b), S_1 is successful synthesis and the imperative semantics enforces to make the b -labeled transition non-possible and to apply the imperative semantics in order to enforce ϕ' being true in the c -labeled discrete successor. The figure 6(b) shows a fragment of the (infinite) semantic model $\mathcal{M}_1^S$ of a DMUS-T model $\mathcal{M}_1$; and the fragment of the updated semantic model $\mathcal{M}_2^S$. However, remark that the incremental synthesis is not computed over the semantic model $\mathcal{M}_1^S$ (infinite state-space), but over a finite quotient of $\mathcal{M}_1^S$ (see section 7). The actions performed in the finite quotient are translated into a new DMUS-T model $\mathcal{M}_2$ in which, for instance, $\Psi_{np}(q, b)$ is updated. The new DMUS-T model is such that the knowledge level of a requirement I_R in $\mathcal{M}_2$ is not lower than in $\mathcal{M}_1$ (see lemma 2 in section 5 for the proof), that is, $\models_{\mathcal{M}_1} I_R \leq_k \models_{\mathcal{M}_2} I_R$ (footnote ⁴).

In the figure 7, the synthesis rule S_2 is applied to the state $s = (q, \gamma, c_s)$, a synthesis context for S_2 . Synthesizing *true* in every time successor of s which satisfies the premise ($y \leq 5$) enforces turning into possible the unspecified τ -transitions

⁴ $\models_{\mathcal{M}} I_R$ is the truth value $\models (\phi, (q_0, \gamma^0, 1))$ where $(q_0, \gamma^0, 1)$ is the initial state of the model $\mathcal{M}$ with an accessibility condition 1.

leading to a state satisfying $(y \leq 5)$. On the other hand, synthesizing *false* enforces turning into non-possible all the unspecified τ -transitions which lead to a state satisfying $(y > 5)$. The actions performed in the abstract model are again translated to a new DMUS-T model and, for instance, $I_p(q)$, $I_{np}(q)$ are updated.

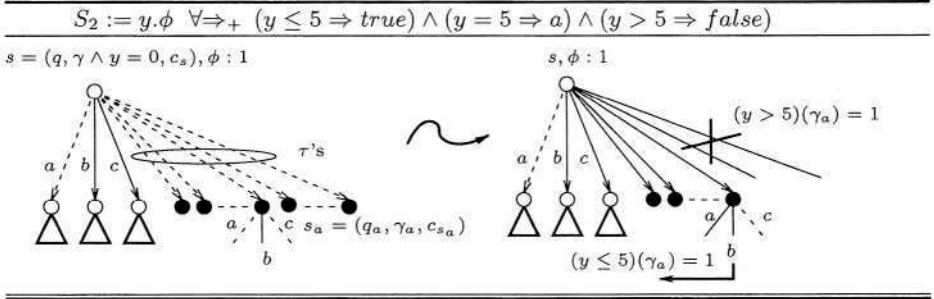


Fig. 7. Imperative Semantics (updating the invariants in a DMUS-T model): S_2 is successful synthesis and the actions performed in the abstract model are shown

7 Implementation

A prototype system based on the approach described in this paper has been implemented and initial results are reported in the conclusions. Some aspects of the implementation of the prototype tool are discussed in the following paragraphs.

Incremental Synthesis algorithm. Regarding the synthesis algorithm, unfortunately, tableau construction for dense real-time logics allowing punctuality is undecidable [5]. Incremental synthesis applies a bounded model construction algorithm similar to the one in [12]. In the bounded synthesis approach, given a SCTL-T^s scenario and a source DMUS-T model, a satisfying target model is synthesized (if feasible) within given bounds on the number of clocks in $\mathcal{C}$ and constants in clock constraints. Given this bounds the synthesis algorithm is computed over a finite quotient as it is described below.

Obtaining a finite abstraction. SCTL/MUS-T is a dense-time methodology, so automation of model checking and synthesis involves obtaining an exact and finite abstraction of the infinite and dense state-space. In this respect, two many-valued strong time abstracting bisimulations (FSTaB and BSTaB) are defined in [6] which preserve truth values in SCTL-T. Model checking and incremental synthesis algorithms use these STaBs to compute such finite and exact quotient by using a minimization algorithm similar to the one in [13]. In this way, we can reduce the model checking and the incremental synthesis to the untimed multi-valued case. So, the existing tool for the untimed methodology SCTL/MUS [9] can be exploit only by adding the logic for reasoning about time propositions.

8 Related Work

Most of the formal techniques require the specification to be complete in some sense before the analysis can start. In [14], incomplete specifications are supported by using the real-time process calculi TMS (Timed Modal Specifications). TMS introduce two modalities in transitions, *may* and *must*. Despite this characterisation supports refinements in an incremental process, consistency checking is not implicit since *not-must* transitions are not considered. On the other hand, the approach in LSCs (Live Sequence Charts) [15,16] allows to specify not only possible behavior, but also mandatory and forbidden ones (anti-scenarios). However, the far or the close a TM or a LSC specification is from satisfying a property-oriented specification is not measured by means of many-valued reasoning. The exploitation of imperative semantics in order to provide constructive methods for building correct specifications in a systematic way is almost exceptional. Purely imperative uses of logic in computer science has been pursued for untimed systems in several works (see [17] for a collection). Under a suitable interpretation, one may view any resolution mechanism, for instance satisfiability, as model building but, in dense-time logics, the satisfiability problem is restricted by undecidable results when punctuality is allowed [5].

The incremental synthesis of a SCTL-T^s scenario can be viewed as a form of the more traditional controller synthesis problem where the specified transitions (possible and non-possible) are uncontrollable and the unspecified ones are controllable. In this sense the timed version of the controller synthesis problem [18] could be applied by handling the unspecified transitions, that is, turning unspecified transitions into possible or non-possible ones. However, a MUS-T model is really a multi-valued model with an arbitrary number of clocks, in which only the clocks in $\mathcal{C}$ are specified, so the controller synthesis solution cannot be applied.

To effectively reason about models which are incomplete in some sense or contain inconsistencies, multi-valued temporal logic model checking has been recently introduced in several works [4,19,20]. However none of these works allows to reason about time, and so, they cannot be applied to models in which uncertainty and disagreement arise in presence of time dependence.

9 Conclusions and Further Work

With respect to other formal approaches proposed in the literature, SCTL/MUS-T is based on an iterative and incremental structure in which real-time characteristics are considered from the beginning. One advantage of this approach is early detection of timing failures. Also, as the system gradually takes form as more is learned about the problem, alternative solutions can be explored.

Regarding the specification process, scenarios are often easier to identify at the beginning than generic requirements that can be made clear only after a deeper knowledge about the system has been gained. While scenarios are an effective means for eliciting requirements, they cannot replace such requirements since they are partial and often leave the underlying requirements implicit. The application of SCTL/MUS-T to the steam-boiler case study [7] has revealed

that specially at early phases: requirements often change guided by verification results; new scenarios are discovered throughout the process; and misunderstanding of the system is uncovered by conflicts in requirements or scenarios. A more coupled relation between requirements and scenarios is under way to enable inferring general requirements from partial scenarios as proposed in [21].

Since our purpose is reconciling formal methods with software practice, more effort is needed in order to reduce the formal background of practitioners. As it is known, temporal logic formulas, especially explicit clock ones, are complex to express and interpret. Meanwhile, practitioners are used to work with documents, very often in natural language, and with graphic notations. We are working on improving the specification process (1) by defining a SCTL-T diagrammatic counterpart similar to the ones in [15,16,22]; and (2) by providing a SCTL-T repository where patterns define parametrisable SCTL-T formulas accompanied by examples of use and documentation in natural language.

Finally, we are working on distributing SCTL/MUS-T for a collaborative specification [23] where multiple perspectives, from multiple stakeholders, are maintained separately as viewpoints. We focus on inconsistency management and on reasoning in presence of it to determine how inconsistencies affect critical requirements. So, a conflict has not to be resolved immediately, but ameliorated or deferred until a better understanding makes the resolution possible.

References

1. Larsen, K.G., Steffen, B., Weise, C.: Continuous Modeling of Real-Time and Hybrid Systems: From Concepts to Tools. *Intl. Journal on Software Tools for Technology Transfer* **1** (1997) 64–85
2. Alur, R.: Techniques for Automatic Verification of Real-Time Systems. PhD thesis, Dept. of Computer Science, Stanford University (1991)
3. van Lamsweerde, A.: Formal Specification: a Roadmap. In: 22nd Intl. Conference on Software Engineering (ICSE'00). ACM Press (2000) 147–159.
4. Chechik, M., Devereux, B., Easterbrook, S.M., Lai, A., Petrovykh, V.: Efficient Multiple-Valued Model-Checking Using Lattice Representations. In: Intl. Conference on Concurrency Theory (CONCUR'01). Volume 2154 of LNCS. Springer (2001) 441–455
5. Alur, R., Henzinger, T.A.: Logics and Models of Real Time: A Survey. In: *Real Time: Theory and Practice*. Volume 600 of LNCS. Springer (1992) 74–106
6. Fernández Vilas, A., Pazos Arias, J.J., Díaz Redondo, R.P.: Extending Timed Automaton and Real-time Logic to Many-valued Reasoning. In: 7th Intl. Symposium on Formal Techniques in Real-Time and Fault-tolerant Systems (FTRTFT'02). Volume 2469 of LNCS. Springer (2002) 185–204
7. Fernández Vilas, A., Pazos Arias, J.J., Gil Solla, A., Díaz Redondo, R.P., García Duque, J., Barragáns Martínez, A.B.: Incremental Specification with SCTL/MUS-T: a Case Study. *The Journal of Systems & Software* **71**(2) (2004) (To appear).
8. Alur, R., Courcoubetis, C., Dill, D.: Model Checking in Dense Real-time. *Information and Computation* **104** (1993) 2–34
9. Pazos Arias, J.J., García Duque, J.: SCTL-MUS: A Formal Methodology for Software Development of Distributed Systems. A Case Study. *Formal Aspects of Computing* **13** (2001) 50–91

10. Barringer, H., Fisher, M., Gabbay, D., Gough, G., Owens, R.: METATEM: An Introduction. *Formal Aspects of Computing* **7** (1995) 533–549
11. Merz, S.: Efficiently Executable Temporal Logic Programs. In: *Executable Modal and Temporal Logics*. Volume 897 of LNCS. Springer (1995) 69–85
12. Laroussinie, F., Larsen, K.G., Weise, C.: From Timed Automata to Logic - And Back. In: *20th Intl. Symposium on Mathematical Foundations of Computer Science (MFCS'95)*. Volume 969 of LNCS. Springer (1995) 529–539
13. Tripakis, S., Yovine, S.: Analysis of Timed Systems using Time-abtracting Bisimulations. *Formal Methods in System Design* **18** (2001) 25–68
14. Cerans, K., Godskesen, J.C., Larsen, K.G.: Timed Modal Specifications – Theory and Tools. In: *5th Intl. Conference on Computer Aided Verification (CAV'93)*. Volume 697 of LNCS. Springer (1993) 253–267
15. Harel, D., Kugler, H.: Synthesizing Sate-based Object Systems from LSC Specifications. *Intl. Journal of Foundations of Computer Science* **13** (2002) 5–51
16. Harel, D., Marelly, R.: Playing with Time: On the Specification and Execution of Time-Enriched LSCs. In: *10th Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'02)*. IEEE Computer Society Press (2002) 193–202
17. Fisher, M., Owens, R., eds.: *Executable Modal and Temporal Logics*. Volume 897 of LNCS. Springer (1995)
18. Altisen, K., Goessler, G., Sifakis, J.: Scheduler Modeling Based on the Controller Synthesis Paradigm. *Real-Time Systems* **23** (2002) 55–84
19. Bruns, G., Godefroid, P.: Model Checking Partial State Spaces with 3-valued Temporal Logics. In: *11th Intl. Workshop on Computer Aided Verification (CAV'99)*. Volume 1633 of LNCS. Springer (1999) 274–287
20. Konikowska, B., Penczek, W.: Model Checking for Multi-Valued CTL*. In: *Beyond Two: Theory and Applications of Multiple-valued Logic*. Springer (2003) 193–210
21. van Lamsweerde, A., Willemet, L.: Inferring Declarative Requirements Specifications from Operational Scenarios. *IEEE Transactions on Software Engineering, Special Issue on Scenario Management* **24** (1998) 1089–1114
22. Moser, L.E., Ramakrishna, Y.S., Kutty, G., Melliari Smith, P.M., Dillon, L.K.: A Graphical Environment for Design of Concurrent Real-Time Systems. *ACM Transactions on Software Engineering and Methodology* **6** (1997) 31–79
23. Barragáns Martínez, A.B., García Duque, J., Pazos Arias, J.J., Fernández Vilas, A., Díaz Redondo, R.P.: Requirements Specification Evolution in a Multi-Perspective Environment. In: *26th Annual Intl. Conference on Computer Software and Applications (COMPSAC'02)*. IEEE Computer Society Press (2002) 39–44

Integrating Temporal Logics

Yifeng Chen and Zhiming Liu

Department of Computer Science,
University of Leicester, Leicester LE1 7RH, UK
{Y.Chen, Z.Liu}@mcs.le.ac.uk

Abstract. In this paper, we study the predicative semantics of different temporal logics and the relationships between them. We use a notation called *generic composition* to simplify the manipulation of predicates. The modalities of possibility and necessity become generic composition and its inverse of converse respectively. The relationships between different temporal logics are also characterised as such modalities. Formal reasoning is carried out at the level of predicative semantics and supported by the higher-level laws of generic composition and its inverse. Various temporal domains are unified under a notion called *resource cumulation*. Temporal logics based on these temporal domains can be readily defined, and their axioms identified. The formalism provides a framework in which human experience about system development can be formalised as refinement laws. The approach is demonstrated in the transformation from Duration Calculus to Temporal Logic of Actions. A number of common design patterns are studied. The refinement laws identified are then applied to the case study of water pump controlling.

1 Introduction

A propositional modal logic consists of a pair of operators describing possibility and necessity in addition to propositional logic [1]. Each collection of axioms determines an axiomatic system. Temporal logics are special modal logics with additional axioms (e.g. the axiom of transitivity).

Most traditional studies tend to focus on the theoretical properties of individual axiomatic systems with a single pair (or very few pairs) of modalities. This has become increasingly inadequate for more sophisticated applications that emerged in recent years. An important application area is the development of hybrid systems. A hybrid system consists of both continuous components that observe continuous physical laws and discrete components that execute digital instructions. Hybrid systems inevitably involve time as an observable and can be naturally specified using temporal logics. Different temporal logics tend to emphasise different aspects of a hybrid system. For example, interval logics such as Duration Calculus (DC) [16], emphasising properties over an interval, are more suitable for describing high-level continuous properties and hence closer to the continuous aspects of hybrid systems. On the other hand, Linear Temporal Logics (LTL) [10], emphasising the properties of states at discrete time points, are more suitable for modelling discrete aspects of hybrid systems and can be easily

verified with timed automata [11]. A straightforward specification in one logic may become less intuitive in another logic. In the past, all aspects of a hybrid system were normally specified in one logic [8,17]. The traditional method of combining logics is to collect all syntactical constructs together and identify the axioms of the system. This usually results in a complicated axiomatic system that is difficult to handle. For example, the design of a hybrid system may start as an abstract specification of the requirements in DC and then become refined by a concrete LTL specification that describes the system behaviour of an implementation. Existing design techniques do not support such a refinement. A natural solution is to interpret different logics at a common semantic level so that the relationships between the logics can be systematically studied.

Predicative interpretation is a standard technique in modal logic [1,14]. A proposition with modal operators can be interpreted as a predicate. The modality of possibility (or necessity) is represented as an existential (or universal) quantifier.

Predicates are also used in semantic modelling of programming languages. This approach is often known as *predicative semantics* [5,7]. A program can be represented as a predicate. Combinators of programs become operators on predicates. In this paper, we will interpret modal/temporal logics using predicative semantics and reason about the relationships between them at this level.

Predicative semantics is observation-based. A predicate can be interpreted as a set of possible observations on the observables (i.e. logical variables). All common combinators are relational in observation-based semantics (i.e. they distribute over universal disjunction). In order to manipulate predicates and their operators flexibly at a higher level of abstraction, we use a notation called *generic composition* [2]. A generic composition is a relational composition with a designated interface consisting of several logical variables. A specification in real applications may involve complicated constraints of several temporal logics. Each temporal logic emphasises a particular observable aspect of the system. This is why generic composition with a restricted interface is more convenient than simple relational composition. Generic composition has an inverse operator. With the help of the two operators, we no longer need the existential and universal quantifiers. The modality of possibility then becomes a generic composition, while the modality of necessity becomes its inverse of converse. The modality of ‘until’ can be defined as a combination of generic composition and its inverse. Temporal logics have been characterised algebraically using Galois connections [15]. In our approach, the accessibility relation of modal logics is directly parameterised. It is hence possible to study logics with different temporal domains in a same semantic framework.

The link between two specifications in different temporal logics can be characterised as a pointwise relation between the possible observations of the specifications. Such a pointwise relation also determines a pair of modalities and can be defined with a generic composition and its inverse.

For the purpose of unification, we model different temporal domains such as real time, traces, timed traces, forests (for branching time) and intervals using a notion called *resource cumulator* [3]. A cumulator is a quintuple comprising a monoid, a corresponding partial order and a volume function that measures

the amount of resources. A cumulator provides the ‘type’ for a logical variable of a particular temporal domain.

The integration of different temporal logics will not be useful unless we provide the knowledge about how specifications in one logic can be approximated or refined by specifications in another logic. Such knowledge can be formalised as the refinement laws of modalities. Identifying these laws can make the design process more systematic. In this paper, we will demonstrate this by studying the refinement from DC specifications to TLA implementations. As we explained before, DC is natural in describing high-level duration properties of the continuous part of a hybrid system. Schenke and Olderog [13] studied the direct refinement transformation from DC to a language similar to CSP [6]. Since the gap between DC and TLA specifications is smaller than that between DC and a real programming language, our approach yields stronger algebraic properties. The result TLA implementations can be verified with model-checking tools.

Section 2 studies the predicative semantics of modal logic using the notation of generic composition and its inverse. Section 3 unifies different temporal domains under the notion of resource cumulator and defines the predicative semantics of temporal logic in general. Section 4 discusses several well-known temporal logics. The relationships between them are studied in Section 5. The refinement laws identified in Section 5 are then applied to the case study in Section 6.

2 Predicative Semantics of Modal Logics

Manipulating Predicates

We assume that there are two types of logical variables: non-overlined variables such as $x, y, z, \dots$ and overlined variables such as $\bar{x}, \bar{y}, \bar{z}, \dots$. Overlining is only used to associate corresponding logical variables syntactically. We use a notation called *generic composition* [2] to manipulate predicates. A generic composition is a relational composition with a designated interface of non-overlined variables.

Def 1 $P :_x R \triangleq \exists x_0 \cdot P[x_0/x] \wedge R[x_0/\bar{x}]$.

A ‘fresh’ variable x_0 is used to connect x of P and $\bar{x}$ of R and hidden by the existential quantifier. Generic composition is a restricted form of relational composition. It relates two predicates on only some of their logical variables. For example, the following composition relates two predicates on only x (and $\bar{x}$):

$$(x=10 \wedge y=20) :_x (\bar{x} \leq x \wedge z=30) = (10 \leq x \wedge y=20 \wedge z=30).$$

The existential quantifier $\exists x \cdot P$ is simply represented as $P :_x \text{true}$, and variable substitution $P[e/x]$ as $P :_x (\bar{x}=e)$. An interface x may split into several variables, e.g. (y, z) . For example, the generic composition $P :_{(y, z)} \text{true}$ is the same as the predicate $\exists y \exists z \cdot P$. If the vector is empty, a generic composition becomes a conjunction: $P : R = P \wedge R$.

Generic composition has an inverse operator denoted by $P /_x R$, which is the weakest predicate X such that $(X :_x R) \subseteq P$. It can be defined by a Galois connection:

Def 2 $X \subseteq P /_x R$ iff $X :_x R \subseteq P$ for any predicate X .

Generic composition and its inverse satisfy a property:

$$P /_x R = \neg(\neg P :_x \tilde{R}) = \forall z \cdot (R[z/x, x/\bar{x}] \Rightarrow P[z/x])$$

where $\tilde{R} \hat{=} R[\bar{x}/x, x/\bar{x}]$ is the converse of R for the variable x . Universal quantifier $\forall x \cdot P$ can then be written as $P /_x \text{true}$. Negation $\neg P$ becomes false / P whose interface is empty. Implication $P \Rightarrow Q$ becomes Q / P with an empty interface. Disjunction $P \vee Q$ is a trivial combination of negation and implication. Thus all connectives, substitution and quantifiers become special cases of generic composition and its inverse [2].

Theorem 1 *Generic composition and its inverse are complete in the sense that any predicate that does not contain overlined/free variables can be written in terms of generic composition and its inverse using only the constant predicates and predicate letters.*

The theorem shows the expressiveness of generic composition for predicate manipulation. Generic composition and its inverse form a Galois connection and satisfy the algebraic laws of strictness, distributivity and associativity.

Law 1

(1) $A \subseteq (A :_x R) /_x R$	(2) $(A /_x R) :_x R \subseteq A$
(3) $\text{false} :_x R = \text{false}$	(4) $\text{true} /_x R = \text{true}$
(5) $A :_x (R \vee S) = (A :_x R) \vee (A :_x S)$	(6) $(A \vee B) :_x R = (A :_x R) \vee (B :_x R)$
(7) $A /_x (R \vee S) = (A /_x R) \wedge (A /_x S)$	(8) $(A \wedge B) /_x R = (A /_x R) \wedge (B /_x R)$
(9) $(A :_x R) :_x S = A :_x (R :_x S)$	(10) $(A /_x R) /_x S = A /_x (S :_x R)$

The notation is especially useful when the interfaces of the operators in a predicate are not identical. In the following laws we assume that x, y and z are three different logical variables, $A = \exists z \cdot A$ (independence of the variable z) and $C = \exists \bar{y} \cdot C$ (independence of the variable $\bar{y}$).

Law 2

- (1) $A :_x C = A :_{(x,y)} (\bar{y} = y \wedge C)$
- (2) $A /_x C = A /_{(x,y)} (\bar{y} = y \wedge C)$
- (3) $(A :_{(y,x)} B) :_{(x,z)} C = A :_{(y,x)} (B :_{(x,z)} C)$

In this paper, we will use generic composition and its inverse to define modalities. These properties make the composition a useful technical tool for linking temporal logics. Generic composition has also been applied to define a variety of healthiness conditions and parallel compositions. The above laws and a series of other laws can be found in [2].

Interpreting Modalities

Under Kripke semantics [1], modal logics are logical systems of relations (called “accessibility relations”). Here, we represent a specification as a predicate on a

modal variable (e.g. x) and an *auxiliary variable* (e.g. y). The modal variable records the observable aspect related to the accessibility of the modalities, while the auxiliary variable records the unrelated observable aspect. For now, the variables are left untyped. These logical variables will later be typed in temporal logics. A logical variable may split into several ones, and its type becomes the product of several types. The semantic space is the set of all such specifications (e.g. denoted by $\mathcal{A}$). An accessibility relation $R = R(\bar{x}, x)$ is denoted by a predicate on two variables: the modal variable x and the overlined modal variable $\bar{x}$. Overlined variables only appear in the accessibility relations. Each accessibility relation determines a pair of modalities.

Def 3 $\Diamond_{\mathcal{A}} P \triangleq P :_x \tilde{R}$ and $\Box_{\mathcal{A}} P \triangleq P /_x R$.

The operator $\Diamond_{\mathcal{A}} P$ informally means that “the predicate P may be true” and is defined as a generic composition of the specification P and the converse relation $\tilde{R}$; its dual modality $\Box_{\mathcal{A}} P$ informally means that “the predicate P must be true” is defined with an inverse operator.

If we replace the accessibility relation with its converse, we will obtain a pair of converse modalities.

Def 4 $\Diamond_{\mathcal{A}} P \triangleq P :_x R$ and $\Box_{\mathcal{A}} P \triangleq P /_x \tilde{R}$.

Generic composition and its inverse can be regarded as parameterised modal operators. They have a designated interface and are more convenient than traditional relational composition in this context for two reasons. Firstly, the observable aspects (described by the auxiliary variable) unrelated to the accessibility relation can be excluded from the interface of the relational composition. Secondly, the predicate on the left-hand side of a generic composition (or its inverse) can be either a specification (without overlined variables) or an accessibility relation (with overlined variables). Thus the operators can be directly used to represent the composition of accessibility relations (i.e. the composition of modalities).

The converse/inverse relationships between these modalities are illustrated in a diagram (see Figure 1). The four modalities form two Galois connections.

Law 3 $\Diamond_{\mathcal{A}} P \subseteq Q$ iff $P \subseteq \Box_{\mathcal{A}} Q$ for any $P \in \mathcal{A}$ and $Q \in \mathcal{B}$
 $\tilde{\Diamond}_{\mathcal{A}} P \subseteq Q$ iff $P \subseteq \Box_{\mathcal{A}} Q$ for any $P \in \mathcal{A}$ and $Q \in \mathcal{B}$.

Another important modal operator is ‘until’. $P \mathcal{U}_{\mathcal{A}} Q$ informally means that “ Q may be true, and P must hold at least up to the happening of Q ”.

Def 5 $P \mathcal{U}_{\mathcal{A}} Q \triangleq Q :_x (\tilde{R} \wedge (P(x') /_{x'} (R(\bar{x}, x') \wedge R(x', x))))$.

Here we assume that the accessibility relation is transitive, i.e. $R :_x R \subseteq R$, and the variable x' is fresh and free of naming conflict.

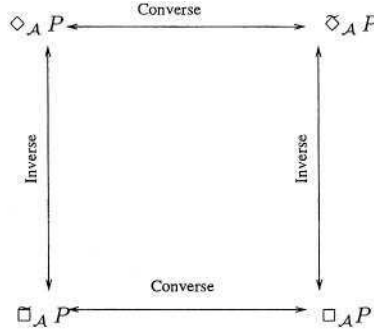


Fig. 1. Diagram of converse/inverse relationships

Transformer Modalities

The transformation between two temporal logics also becomes modalities. Let $\mathcal{A}$ (or $\mathcal{B}$) be a semantic space of specifications, each of which is a predicate on modal variable x (or x') and auxiliary variable y (or y'). The transformation from $\mathcal{A}$ to $\mathcal{B}$ is characterised as a transformation predicate $T = T(\bar{x}, \bar{y}, x', y')$ on four variables. The predicate determines a transformer modality $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}}$ from $\mathcal{A}$ to $\mathcal{B}$ and a corresponding inverse transformer $\Box_{\mathcal{B} \rightarrow \mathcal{A}}$ from $\mathcal{B}$ to $\mathcal{A}$. In the following definition, we assume that $P = P(x, y)$ and $Q = Q(x', y')$.

Def 6 $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P \triangleq P :_{(x, y)} T$
 $\Box_{\mathcal{B} \rightarrow \mathcal{A}} Q \triangleq Q /_{(x', y')} T.$

Note that $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}}$ and $\Box_{\mathcal{B} \rightarrow \mathcal{A}}$ form just one pair of transformers based on the predicate T . Other transformers between the two logics can be denoted as $\Diamond_{\mathcal{A} \rightarrow' \mathcal{B}}$ and $\Diamond_{\mathcal{A} \rightarrow'' \mathcal{B}}$ etc. Let $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}}$ and $\Diamond_{\mathcal{B} \rightarrow \mathcal{C}}$ be two transformers. Their composition $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}} \Diamond_{\mathcal{B} \rightarrow \mathcal{C}}$ is also a transformer (from $\mathcal{A}$ to $\mathcal{C}$), so is the composition of their inverses.

We now identify some of the laws that will be used in our later case studies. The laws can be routinely proved. A transformer and its inverse form a Galois connection and therefore satisfies the following laws.

Law 4 (1) $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P \subseteq Q$ iff $P \subseteq \Box_{\mathcal{B} \rightarrow \mathcal{A}} Q$ (for any P and Q)
 (2) $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}} \Box_{\mathcal{B} \rightarrow \mathcal{A}} \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P = \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P$
 (3) $\Box_{\mathcal{B} \rightarrow \mathcal{A}} \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} \Box_{\mathcal{B} \rightarrow \mathcal{A}} Q = \Box_{\mathcal{B} \rightarrow \mathcal{A}} Q.$

If the transformer predicate $T = (\bar{x} = f(x', y') \wedge \bar{y} = g(x', y'))$ is determined by (possibly partial) surjective functions, the modalities form a Galois embedding, and the transformer $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}}$ distributes conjunction.

Law 5 (1) $\Box_{\mathcal{B} \rightarrow \mathcal{A}} \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P = P$
 (2) $\Diamond_{\mathcal{A} \rightarrow \mathcal{B}} (P \wedge Q) = \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P \wedge \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} Q.$

If the accessibility relations of $\mathcal{A}$ and $\mathcal{B}$ satisfy an additional condition of “monotonicity”: for any y , $R_{\mathcal{A}}(\bar{x}, x) \Rightarrow R_{\mathcal{B}}(f(\bar{x}, y), f(x, y))$, then the transformer and the modalities of necessity become commutative.

Law 6 $\Box_{\mathcal{B}} \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} P = \Diamond_{\mathcal{A} \rightarrow \mathcal{B}} \Box_{\mathcal{A}} P.$

If the modal variable and the auxiliary variable are untyped, the above predicative semantics is contained in predicate calculus and hence complete in the sense that a well-formed formula is always true if and only if it can be proved using the laws of generic composition and its inverse (or equivalently, the axioms of predicate calculus).

3 Temporal Logic of Resource Cumulation

Resource Cumulation

Many aspects of computing can be modelled as the cumulation of resources. In real-time computing, *time* is a kind of resource. A process “consumes” a non-negative amount of time. A computation may also produce resources. For example, a reactive process generates an increasingly longer sequence of intermediate states called a *trace*. Resource cumulation can be formalized as a quintuple called a *cumulator*. $(X, \leq; 0, \wedge; |\cdot|)$, which consists of three parts: a well-founded partial order $(X, \leq)$ in which each element is called a *cumulation* and the greatest lower bound exists for any non-empty subset, a monoid $(0, \wedge)$ in which 0, or zero cumulation is the least cumulation, and a monotonic and associative binary operation *concatenation* $\wedge$ corresponds to the addition of cumulations, and a monotonic and strict *volume* function $|\cdot|: X \rightarrow [0, \infty]$: We assume that the partial order and the monoid are consistent: $a \leq b \Leftrightarrow \exists c \in X. a \wedge c = b$. The unusual part of a cumulator is the volume function. A volume function measures the amount of resource cumulated. With such additional information we can then reason about the dynamics of resource cumulation. For example, a resource is exhausted when its volume reaches infinity ∞ . The use of volume functions can substantially simplify the reasoning of limit points, continuity, and other topological properties. For a more complete account of resource cumulation, please refer to [3].

Example: The amount of time that a computation consumes can be modelled as a cumulator:

$$RTime \hat{=} ([0, \infty], \leq; 0, +; id)$$

where $+$ is addition. *id* is the identity function.

Example: In some applications, we are interested in temporal properties over a period of time and thus need to reason about temporal intervals. Intervals form a cumulator

$$Interval \hat{=} (I, \leq; \emptyset, \wedge; |\cdot|)$$

where I denotes the set of intervals, each of which is a convex subset i of the real domain $[0, \infty]$ (such that for any $t_1, t_2 \in i$ and $t_3 \in T$, $t_1 \leq t_3 \leq t_2$ implies $t_3 \in i$). For example, $[1, 2]$, $[1, 2)$, $(1, 2]$, $(1, 2)$ and the empty set $\emptyset$ are intervals. Let I denote the set of all intervals. $a \cap b \triangleq a \cup b$ if $a \cap b = \emptyset$, $\sqcup a = \cap b$ and $a \cup b \in I$. The volume of a non-empty interval is its length: $|a| \triangleq \sqcup a - \cap a$ where $\sqcup a$ and $\cap a$ denote the lub and glb of the interval a respectively. The volume of the empty set is zero $|\emptyset| = 0$. The orders $a \leq b$ means that b is a right-hand extension of a , i.e. $\exists c \in I \cdot a \cap c = b$.

Example: Finite and infinite traces form a typical cumulator:

$$\text{Trace}(X) \triangleq (X^\dagger, \leq; \langle \rangle, \wedge; | \cdot |)$$

where X is the type of each element, and $X^\dagger$ the set of all sequences of elements (including the infinite ones). For two sequences $a, b \in X^\dagger$, $a \wedge b$ denotes their concatenation. If a is an infinite sequence, then for any b , $a \wedge b = a$. $a \leq b$ iff a is a *prefix* (i.e. pre-cumulation) of b . $|a|$ denotes the length of a . For example, the length of the empty sequence $\langle \rangle$ is 0. a_i denotes the i -th element of the sequence where $1 \leq i \leq |a|$.

Example: A timed trace is a trace with non-decreasing time stamps. The sequence $\langle (1, p), (2, q), (4, p) \rangle$ is one example. In general, a timed trace is a trace of pairs in the form $\langle (t_0, s_0), (t_1, s_1), \dots, (t_n, s_n), \dots \rangle$. Timed traces form a cumulator:

$$\text{TimedTrace}(X) \triangleq (T(X), \leq; \langle \rangle, \wedge; | \cdot |)$$

where $T(X) \triangleq \{ tr \in ([0, \infty] \times X)^\dagger \mid \forall i, j < |tr| \cdot (i \leq j \Rightarrow t_i \leq t_j) \}$.

Example: A forest is a set of labeled traces, each of which is a tuple consisting of a trace, a start label and an end label. Two labeled traces s and t are connected if the end label of s equals the start label of t . Forests form a cumulator:

$$\text{Forest}(X, L) \triangleq (F(X, L), \leq; \emptyset, \wedge; | \cdot |)$$

where $F(X, L)$ is the set of all forests with the set X of elements and the set L of labels. The concatenation $f_1 \wedge f_2$ between the two forests is the pointwise concatenations of their connected traces together with all non-connected labeled traces. The partial order $\leq$ can be derived from the concatenation operation. The height of a forest is the maximum length of its labeled traces. The height of the empty forest is 0.

Temporal Logic of Resource Cumulation

Temporal logic of resource cumulation is a modal logic. Let $(X, \leq; 0, \wedge; | \cdot |)$ be a cumulator. A general cumulative specification is a predicate on a modal variable $x \in X$ whose type is a cumulator and an untyped auxiliary variable y . We let $\mathcal{R}$ denote the semantic space of such specifications. The general cumulator

gives rise to a number of accessibility relations, each of which determines two pairs of modalities. A common accessibility relation corresponds to the left-hand contractions:

$$R \hat{=} \exists z \in X \cdot (\bar{x} = z \wedge x).$$

The modality $\Diamond_{\mathcal{R}} P$ informally means that “the predicate P becomes true after some pre-cumulation of resources”. More precisely, the behaviours of $\Diamond_{\mathcal{R}} P$ are the behaviours of P extended with arbitrary cumulations on the left-hand side. The modality $\Box_{\mathcal{R}} P$, instead, means that “the predicate P is true for any left-hand extensions of the behaviours of P ”. The pair of converse modalities $\Diamond_{\mathcal{R}} P$ and $\Box_{\mathcal{R}} P$ are actually the corresponding “past-tense” modalities. All properties of general modalities are inherited.”

There exists a dual accessibility relation for right-hand contractions:

$$R' \hat{=} \exists z \in X \cdot (\bar{x} = x \wedge z).$$

Again, it determines two pairs of modalities $\Diamond_{\mathcal{R}'} P$, $\Box_{\mathcal{R}'} P$, $\Diamond_{\mathcal{R}} P$ and $\Box_{\mathcal{R}} P$. The modalities of left-hand and right-hand extensions/contractions commute with each other respectively. Their respective compositions (e.g. $\Diamond_{\mathcal{R}} \Diamond_{\mathcal{R}'} P$) becomes a bi-directional contractions/extensions.

Using the volume function, we can define a variety of restricted modalities. Let c be a condition on real numbers. The condition $c(z) \hat{=} |z| < 3$ is a simple example. The restricted left-hand-contraction relation can be defined:

$$R_c \hat{=} \exists z \in X \cdot (c(z) \wedge \bar{x} = z \wedge x).$$

We shall use $\Diamond_{c(\mathcal{R})} P$ to denote the modality with the accessibility relation R_c (similar for other modalities). $\Diamond_{c(L)} P$ informally means that “the predicate P eventually becomes true after the pre-cumulation of some resources whose volume satisfies the condition c ”. For example, the specification $\Diamond_{|L| < 3} P$ means that “the predicate P eventually becomes true in less than 3 steps”. The ‘next’ operator $\bigcirc P$ becomes $\Diamond_{|L|=1} P$. The most commonly used temporal operator $\Diamond P$ in LTL means that “the predicate P eventually becomes true in finite steps”. Its dual operator $\Box P$ means that “the predicate P is always true after finite steps”. They correspond to $\Diamond_{|L| < \infty} P$ and $\Box_{|L| < \infty} P$ respectively (with the cumulator *Trace*).

In general, temporal logic of resource cumulation is not complete, since the temporal domain (i.e. the cumulator) of the modal variable may not have a complete axiomatic characterisation. Nevertheless, it is still possible to reason about temporal specifications manually based on the semantic properties of the underlying temporal domain.

4 Examples of Temporal Logics

Real Time

The amount of time that a computation consumes corresponds to the cumulator *RTime*. A real-time specification is a predicate on a typed modal variable $t \in$

$[0, \infty]$ that denotes time and an untyped auxiliary variable s that denotes the system's state at the time. We let $\mathcal{T}$ denote the space of such specifications. Since addition is commutative i.e. $a + b = b + a$, it makes no difference whether time is extended from the left-hand side or the right-hand side.

The state variable s can be used to specify various interesting things. For example, a device whose temperature y grows exponentially along time s can be specified as a predicate $(s = e^t)$. There are two different interpretations of this specification: we may interpret each t as an absolute time point and s as the corresponding temperature, or alternatively, we may treat t as the lapse of time and s as a state associated with the time lapse. Real-time logic is not concrete enough to distinguish these two interpretations.

Real-Time Intervals

Intervals within a time domain form the cumulator *Interval*. A specification on intervals is a predicate on a variable $i \in I$ that denotes the interval and an auxiliary variable x that denotes some system feature related to the interval. We let $\mathcal{I}$ denote the space of all temporal specifications on intervals. An interval can be extended from either left-hand side or right-hand side.

Traces, Timed Traces, and Branching Time

Traces of elements of X form a cumulator *Trace*(X). A trace specification is a predicate on a single variable $tr \in X^\dagger$. We let $\mathcal{S}$ denote the space of trace specifications. For example, the specification $\Box_{|S| < \infty} (tr_0 = 1)$ states that the first element of every suffix is 1, i.e. every state is 1. We introduce a dependent variable $s \triangleq tr_0$. The specification can then be simplified as $\Box_{|S| < \infty} (s = 1)$. Such semantic notation directly corresponds to LTL, although here we use a non-standard notation and allow finite traces.

If another dependent variable $s' \triangleq tr_1$ is used to denote the second element of the trace, we can then specify actions. For example, let X be the set of natural numbers. The specification $\Box_{|S| < \infty} (s < s')$ describes a trace of increasing numbers. Temporal Logic of Actions (TLA) [9] is a logic of stuttering-closed specifications on actions. The stuttering closure $[P]_s \triangleq (P \vee s = s')$ of a specification P is a disjunction. For example, the specification $\Box_{|S| < \infty} [s < s']_s$ describes a trace of non-decreasing natural numbers.

The link between the original variables and the dependent variables can also be characterized as a transformer. For example, let $P(s, s')$ be a specification on the current state s and the next state s' . It corresponds to a specification $P_{(s, s')} (\bar{s} = tr_0 \wedge \bar{s}' = tr_1)$ on traces.

Timed traces form the cumulator *TimedTrace*(X). We let $\mathcal{K}$ denote the space of specifications on timed traces. For TLA of timed traces, we introduce dependent variables $s \triangleq s_0$, $s' \triangleq s_1$, $t \triangleq t_0$ and $t' \triangleq t_1$ and assume that $t \leq t'$. The stuttering closure is defined: $[P]_{(s, t)} \triangleq P \vee (s = s' \wedge t = t')$. For example, the following specification requires the state to change from 1 to 0 after no-more-than 4 seconds or from 0 to 1 after no-less-than 26 seconds:

$$\Box_{\mathcal{K}} [(s = 1 \wedge s' = 0 \wedge t' - t \leq 4) \vee (s = 0 \wedge s' = 1 \wedge t' - t \geq 26)]_{(s, t)}. \quad (1)$$

Forests form the cumulator $Forest(X)$. The temporal logic of forests is a logic of branching time. Similar to LTL, we normally use finitely-restricted modalities to describe safety and liveness properties for “all branches”. On the other hand, if the volume function is the minimum length of labeled traces, a forest becomes infinite if any one of its branches reaches infinity. We will then obtain modalities of safety and liveness properties for “some branches” [4].

Duration Calculus

Duration calculus (DC) is a special interval logic. A duration specification is a predicate on a variable $i \in I$ that denotes the interval and an auxiliary variable $x: [0, \infty] \rightarrow \{0, 1\}$ that denotes a real-time Boolean function. The space of duration specifications is denoted by $\mathcal{D}$.

Again, we may introduce some dependent variables. For example, instead of specifying the relation (i.e. a predicate) between the interval and the real-time function, we may specify the relation between the length of the interval and the integral of the real function in the interval. Although not all computation can be specified in such a restricted way, it has been expressive enough for most applications and covers most common design patterns [12]. Here we shall use $t \triangleq |i|$ to denote the length of the interval and $s \triangleq \int_b^e x(t) dt$ to denote the integral of the function in the interval.

For example, the Gas Burner problem [12] includes a requirement that gas leak is bounded by 4 for any interval shorter than 30. This can be formalised as a specification in $\mathcal{D}$:

$$\Box_{\mathcal{D}} (t \leq 30 \Rightarrow s \leq 4) \quad (2)$$

where t and s are two dependent variables.

The following two concrete DC specifications form a common design that implements the above abstract specification:

$$\Box_{\mathcal{D}} ([x] \Rightarrow t \leq 4) \quad \text{and} \quad \Box_{\mathcal{D}} ([x] \ ; \ [\neg x] \ ; \ [x] \Rightarrow t \geq 26) \quad (3)$$

where the real-time function $x(\ell)$ records whether there is gas leak at the time point ℓ , the specification $[x] \triangleq (s=t)$ describes a period with gas leak (at “most” time points of in the period [16]), and $[\neg x] \triangleq (s=0)$ describes a period without leak. The first specification requires any leaking period to be bounded by 4 seconds; the second specification states that, during any interval, the period of non-leak between two periods of leak should be no less than 26 seconds. The sequential composition $;$ (also known as the *chop operation*) is the pointwise concatenation of the intervals of specifications:

$$P \ ; \ Q \triangleq \exists i_1 i_2. (P[i_1/i] \wedge Q[i_2/i] \wedge i = i_1 \hat{\ } i_2).$$

5 Linking Temporal Logics

The examples of last section showed that different temporal logics are good at describing different aspects of a system at different levels of abstraction. The

abstract requirement is naturally specified in DC. The following DC design has described a controller switching between ‘on’ and ‘off’ at particular *time points* (while the state remains “mostly” unchanged between consecutive time points). Although such controlling can be described in DC, the essentially equivalent TLA specification (1) is arguably more intuitive. The challenge here is to link them in the same theoretical framework so that an implementation in TLA can be obtained from an abstract specification in DC through refinement laws. Such laws will become the formal representation of our experience about the development of hybrid systems.

Real Time and Real-Time Intervals

In Section 4, we identified two distinct informal interpretations of a real-time specification. It turns out that they correspond to two different transformers from $\mathcal{T}$ to $\mathcal{I}$.

A real-time specification on absolute time points can be transformed to an interval specification on the left ends of intervals. The transformers satisfy Laws 5 and 6.

$$\begin{aligned} \text{Def 7} \quad \Diamond_{\mathcal{T} \rightarrow \mathcal{I}} P &\triangleq P :_t (\bar{t} = \Box i) \\ \Box_{\mathcal{I} \rightarrow \mathcal{T}} P &\triangleq P /_t (\bar{t} = \Box i). \end{aligned}$$

For example, the specification $x = e^t$ about absolute time t and temperature x is transformed to an interval specification $x = e^{\Box i}$ independent of the right end of interval i . It informally means that “*the temperature grows exponentially at the start of the interval*”.

Many practical systems are independent of absolute starting time. For example, a cash machine must respond to cash withdrawal any time in a day, although there can be timeout restrictions during any operation. Such a service only depends on the lapse of time. It can be specified in real-time logic and then transformed to interval logic.

$$\begin{aligned} \text{Def 8} \quad \Diamond_{\mathcal{T} \rightarrow' \mathcal{I}} P &\triangleq P :_t (\bar{t} = |i|) \\ \Box_{\mathcal{I} \rightarrow' \mathcal{T}} P &\triangleq P /_t (\bar{t} = |i|). \end{aligned}$$

For example, the real-time specification $x = e^t$ is transformed to an interval specification $x = e^{|i|}$ independent of the absolute starting time. The specification informally means that “*the temperature has grown exponentially during the interval*”.

The informally different interpretations real-time specifications now become formally distinguishable in interval logic.

Real-Time Intervals and Duration Calculus

DC is a special kind of real-time interval logic. The link between them can be characterised as the following transformer.

$$\begin{aligned}\text{Def 9} \quad \Diamond_{\mathcal{I} \rightarrow \mathcal{D}} P &\triangleq P \cdot_s (\bar{s} = \int_i x(t) dt) \\ \Box_{\mathcal{D} \rightarrow \mathcal{I}} P &\triangleq P /_s (\bar{s} = \int_i x(t) dt).\end{aligned}$$

Here $s \in \mathbb{R}^+$ represents the integral accumulated during an interval, and we assume that $0 \leq s \leq |i|$. We may also view s as a dependent variable for $\int_i x(t)$. The transformer forms an embedding and therefore satisfies Laws 5 and 6.

Real Time and Duration Calculus

In Section 4, we discussed a typical pattern of duration specifications, each of which is a predicate $P(t, s)$ on the two dependent variables. The link between the dependent variables and the original variables can be characterized as a transformer. Indeed the transformation from real-time specifications to duration specifications is the composition of the transformation from real-time specifications to interval specifications and the transformation from interval specifications to duration specifications.

$$\begin{aligned}\text{Def 10} \quad \Diamond_{\mathcal{T} \rightarrow \mathcal{D}} P &\triangleq \Diamond_{\mathcal{I} \rightarrow \mathcal{D}} \Diamond_{\mathcal{T} \rightarrow \mathcal{I}} P \\ \Box_{\mathcal{D} \rightarrow \mathcal{T}} P &\triangleq \Box_{\mathcal{I} \rightarrow \mathcal{T}} \Box_{\mathcal{D} \rightarrow \mathcal{I}} P.\end{aligned}$$

Here, we are taking the second interpretation of real time (as the length of the interval). Since the length of interval is monotonic in the sense that if $i \subseteq i'$ then $|i| \leq |i'|$, Law 6 of commutativity also holds.

The requirement of Gas Burner in Section 4 can now be formalised as:

$$\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \leq 30 \Rightarrow s \leq 4).$$

A real-time Boolean function $x(\ell)$ satisfies this specification if and only if for any interval, the integral s of $x(\ell)$ during the interval and the length t of the interval satisfy $(t \leq 30 \Rightarrow s \leq 4)$. The above example corresponds to a general specification pattern:

$$\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \leq A \Rightarrow s \leq B) \tag{4}$$

where A and B are constant parameters such that $A \geq B$. This pattern of specification requires a system not to stay in the Boolean state 1 longer than B during any period no longer than A . It has dual pattern that requires a system not to stay in the state 0 for too long but stay in the state 1 long enough: $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \geq A \Rightarrow s \geq B)$. The two patterns are illustrated in Figures 2 as sets of coordinates (t, s) . Note that we always assume $s \leq t$.

Let $f(\ell)$ be a monotonically-nondecreasing function such that $f(\ell) \leq \ell$ for any ℓ . The following specification is a generalisation of pattern (4):

$$\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} s \leq f(t)$$

in which the function sets the least upper bound for s . It is monotonic and non-decreasing as we naturally assume that, for any longer interval, the least upper bound is allowed to be greater. The general pattern has a dual $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} s \geq g(t)$

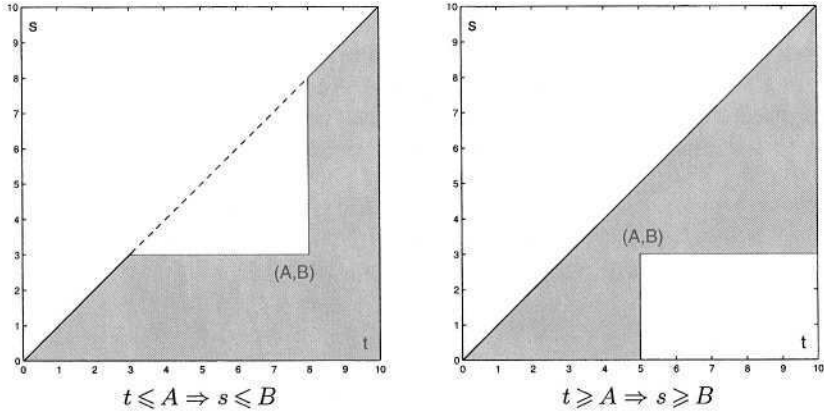


Fig. 2. Basic patterns of DC

where the function g is also monotonic and non-decreasing and satisfies $g(\ell) \leq \ell$ for any ℓ .

The following laws show that the general patterns can be decomposed as the conjunction of basic patterns.

- Law 7**
- (1) $\Box_{\mathcal{T}} s \leq f(t) = \bigwedge_{\ell} \Box_{\mathcal{T}} (t \leq \ell \Rightarrow s \leq f(\ell))$
 - (2) $\Box_{\mathcal{T}} s \geq g(t) = \bigwedge_{\ell} \Box_{\mathcal{T}} (t \geq \ell \Rightarrow s \geq g(\ell))$.

TLA and Duration Calculus

We now study a technique to refine DC specifications with TLA designs. Indeed each timed trace of 0s and 1s determines a real-time Boolean function in $[0, \infty] \rightarrow \{0, 1\}$. For example, the timed trace $\langle (1.0, 0), (2.0, 1), (4.0, 0) \rangle$ corresponds to a Boolean real function whose value is 0 from time 1.0 to time 2.0 when the value becomes 1 until time 4.0. The state between any two consecutive time points is constant. For example, the DC abstract specification (2) can be implemented with a TLA specification of timed traces (1). The TLA design is arguably more intuitive than (2) in DC alone. Such interpretation of a timed trace also directly corresponds to a timed automata.

The link between timed-trace TLA and duration calculus can be characterised as a predicate on timed trace $\overline{tr}$, interval i and real-time Boolean function $x(\ell)$. An interval and a real-time Boolean function are consistent with a given timed trace, if the integral of the function over the interval is equal to the sum of all its sub-intervals during which the timed trace has state 1:

$$T(\overline{tr}, i, x) \hat{=} \sum_{k < |\overline{tr}|} \bar{s}_k \times |[\bar{t}_k, \bar{t}_{k+1}] \cap i| = \int_i x(t) dt.$$

- Def 11**
- $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} P \hat{=} P :_{tr} T$
 - $\Box_{\mathcal{D} \rightarrow \mathcal{K}} P \hat{=} P /_{tr} T.$

The basic pattern $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \leq A \Rightarrow s \leq B)$ can be refined with Law 8(1) in which $\supseteq$ denotes the refinement order such that $P \subseteq Q$ if and only if $P \wedge Q = P$. We let $High \triangleq (s = 1 \wedge s' = 0)$, $Low \triangleq (s = 0 \wedge s' = 1)$ and $\Delta t \triangleq t' - t$. Law 8(2) provides a similar refinement for the dual pattern. The following refinement laws can be proved using the algebraic laws of the generic composition and its inverse and the mathematical properties of the underlying interval and trace domains.

Law 8

- (1) $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \leq A \Rightarrow s \leq B) \supseteq$
 $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} \Box_{\mathcal{K}} [(High \wedge \Delta t \leq B/n) \vee (Low \wedge \Delta t \geq (A-B)/n)]_{(s,t)} \quad (n > 0)$
- (2) $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (t \geq A \Rightarrow s \geq B) \supseteq$
 $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} \Box_{\mathcal{K}} [(High \wedge \Delta t \geq B/n) \vee (Low \wedge \Delta t \leq (A-B)/n)]_{(s,t)} \quad (n > 0).$

These laws allow the frequency of switching to multiply for an integer number n of times and hence are more general than the example TLA refinement (1). We can always replace an integer parameter with a real parameter in the above laws if the result is a further refinement. For example, we may replace the first n on the right-hand side of Law 8(1) with any real number $\lambda \geq n$. The parameters A and B are constant parameters. That means the TLA refinement describes a controller that runs according to an internal timer but does not depend on input from the environment. Figure 3 illustrates the refinement of the basic patterns. The grey areas indicate the requirements, while the dark areas (contained in the grey areas) illustrate the TLA designs.

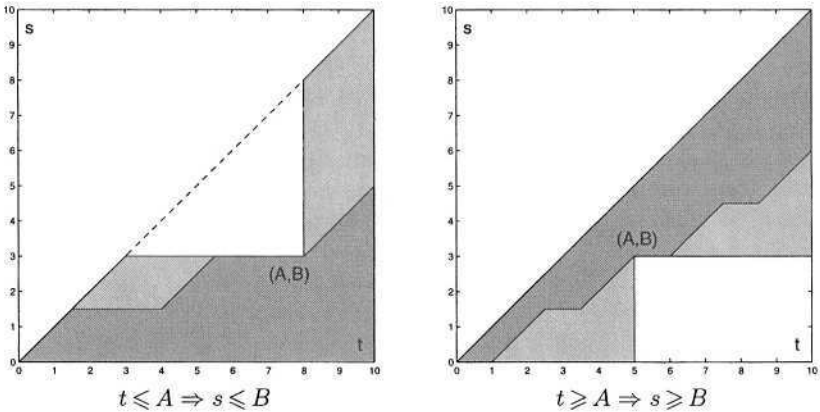


Fig. 3. Refinement of basic patterns ($n = 2$)

The refinement of the general patterns $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} t \leq f(t)$ and $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} t \geq g(t)$ is based on the refinement of the basic patterns (see Figure 4).

Law 9

$$\Diamond_{T \rightarrow D} \Box_T s \leq f(t) \supseteq$$

$$\Diamond_{K \rightarrow D} \Box_K \left[(High \wedge \Delta t \leq a) \vee \left(Low \wedge \Delta t \geq \sup_{\ell \geq a} \frac{\ell - f(\ell)}{\lfloor f(\ell)/a \rfloor} \right) \right]_{(s,t)}$$

where $a > 0$, and for any $\ell \leq a$, $f(\ell) = \ell$.

Law 10

$$\Diamond_{T \rightarrow D} \Box_T g(t) \leq s \supseteq$$

$$\Diamond_{K \rightarrow D} \Box_K \left[\left(High \wedge \Delta t \geq \sup_{\ell \geq b} \frac{g(\ell)}{\lfloor (\ell - g(\ell))/b \rfloor} \right) \vee (Low \wedge \Delta t \leq b) \right]_{(s,t)}$$

where $0 < b \leq \inf_{\ell} (\ell - g(\ell))$.

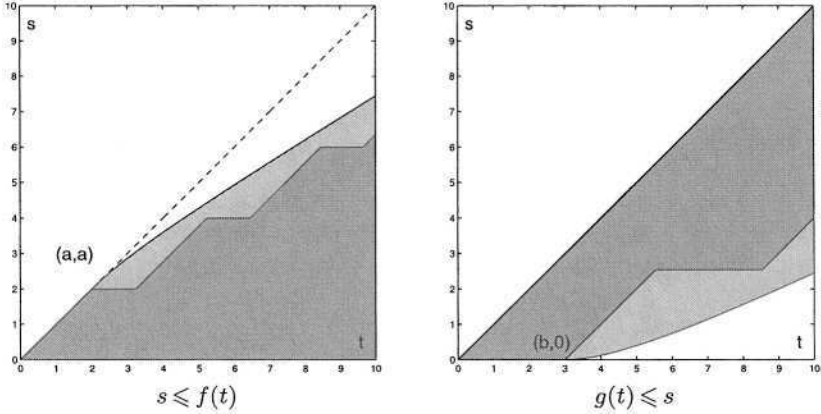


Fig. 4. Refinement of general patterns

In the above refinement laws, we have restricted ourselves to trace-based implementation without input. To incorporate input information from the environment, we assume that the controller not only has an internal timer but is also equipped with a sensor to detect the changes of the environment periodically. If the reading h of the sensor is higher than a particular level H , the switch will be turned on; if the reading is lower than a level L , the switch is off; otherwise, when the level is between H and L , the switch can be either on or off. Let $f(t, h)$ and $g(t, h)$ be monotonic functions with regard to both t and h . The controller periodically checks the input. The (non-zero) cycle of sampling can be as small as possible but must be bounded by a constant τ ; otherwise the controller may not be able to react in time. The following law refines such specifications to the target system.

Law 11 $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} g(t, h) \leq s \leq f(t, h) \supseteq$
 $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} \Box_{\mathcal{K}} [(h \geq H \Rightarrow s = 1) \wedge (h \leq L \Rightarrow s = 0) \wedge \Delta t \leq \tau]_{(s, t)}$
 where $L \leq H$, $\tau \leq f(\tau, L)$ and $g(\tau, H) \leq 0$.

If the functions are linear, we can determine the parameters more accurately. The least upper bound of τ can be determined when assuming $H = L$. Once a particular τ is chosen, the ranges of H and L can be derived.

Law 12 $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (a_1 t - b_1 \leq s - h \leq a_2 t + b_2) \supseteq$
 $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} \Box_{\mathcal{K}} [(h \geq H \Rightarrow s = 1) \wedge (h \leq L \Rightarrow s = 0) \wedge \Delta t \leq \tau]_{(s, t)}$
 where $\tau \leq \frac{b_1 + b_2}{1 + a_1 - a_2}$, $H \leq b_1 - a_1 \tau$ and $L \geq (1 - a_2) \tau - b_2$.

6 Case Study: Water Pump Controlling

To demonstrate the use of the refinement laws of the last section, we first consider an example of a simple water pump. The (hybrid) system consists of a water pool with inflow v_i (which rate is at least v_{imin}) and a water pump. When the water pump is on, water drains at a rate of $v_o > v_{imin}$; when it is off, there is no draining caused by the pump. The requirement is that during any period, the water level never drops more than $\Delta H > 0$.

We assume that the controller has no sensor that can detect the change of inflow or the water level. Under the worst case when v_i is constantly at the lowest rate v_{imin} , the requirement can be specified formally as follows:

$$\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} (v_o s - v_{imin} t \leq \Delta H).$$

It is implicit that s be always bounded by t . Thus we obtain a specification $\Diamond_{\mathcal{T} \rightarrow \mathcal{D}} \Box_{\mathcal{T}} s \leq f(t)$ where a is chosen as its maximum: $a \triangleq \frac{\Delta H}{v_o - v_{imin}}$ and

$$f(t) \triangleq \begin{cases} t & t \leq a \\ v_{imin} t / v_o + \Delta H / v_o & t > a \end{cases}$$

To determine $\sup_{\ell \geq a} \frac{\ell - f(\ell)}{\lfloor f(\ell)/a \rfloor}$, we let $t_n \triangleq a + \frac{n v_o a}{v_{imin}}$ where $n > 0$. The value of $\frac{\ell - f(\ell)}{\lfloor f(\ell)/a \rfloor}$ reaches maximum when ℓ approaches every t_n from its left-hand side. Thus

$$b \triangleq \sup_{\ell \geq a} \frac{\ell - f(\ell)}{\lfloor f(\ell)/a \rfloor} = \frac{t_1 - f(t_1)}{f(t_1)/a - 1} = \frac{v_o^2 a^2}{v_{imin}^2 a + v_m \Delta H} - a$$

The obtained TLA implementation $\Diamond_{\mathcal{K} \rightarrow \mathcal{D}} \Box_{\mathcal{K}} [(High \wedge \Delta t \leq a) \vee (Low \wedge \Delta t \geq b)]_{(s, t)}$ (according to Law 9) is illustrated in Figure 5.

The above example can be generalised in several ways. Firstly, we may require the water level not to drop more than a certain level *within only the intervals shorter than a given constant* t_0 (instead of being in every interval). To refine such a weaker requirement, we need to revise the least-upper-bound function

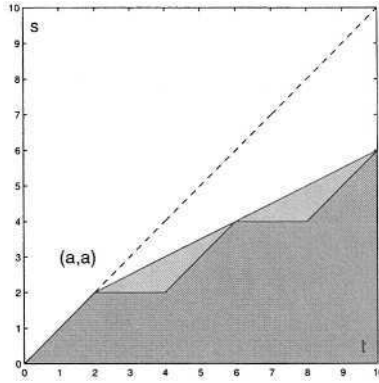


Fig. 5. Controlling of a simple water pump

$f(t)$ slightly. Secondly, if the water level is also required not to rise a certain level, Law 10 can be used for refinement. Since all these modalities distribute conjunction. The two TLA refinements can be combined together in conjunction compositionally. Finally, we have assumed v_{imin} to be a constant (Law 5). If the inflow is not random and fits into some model, then the least average inflow v_{imin} will be a function related to the interval length t . For example, the minimum of $\sin(x)$ is -1 , but its least average for the interval $(-2\pi, 2\pi)$ is 0. If the function $v_{imin}(t)$ is known, then Law 11 is still applicable. This can be generalised further: since we know the relation between the amount of water pumped out (i.e. $v_o s$) and the length t of any corresponding interval, if the pumped water drains into another pool, we can then study the water-level controlling of the other pool using the same refinement laws.

7 Conclusions

This paper has presented a predicative interpretation for modal logics. The accessibility relation of Kripke semantics is parameterised as a predicate. Introducing a new pair of modalities is the same as introducing a new accessibility relation. The transformers between modal logics also become modalities. Formal reasoning is mostly conducted at the level of predicate calculus and assisted with the higher-level laws of generic composition and its inverse. The completeness of the semantic interpretation relies on the completeness of predicate calculus, if the variables are untyped. The reasoning about predicative semantics with typed variables depends on the formalisation of the underlying state space (for the auxiliary variable) and the cumulator (for the modal variable) and may not be complete.

The framework allows us to formalise our knowledge about the relationships between different temporal logics in the form of algebraic or refinement laws. In the case study on DC and TLA, we have identified refinement laws for several

design patterns. Some of the laws are general and cover most types of refinement with a particular target implementation. More specific laws are introduced for the most common patterns, and their parameters can be more readily determined. The technique is applied to the design water-pump controlling system. It is not a trivial task to identify general but at the same time practically useful laws. However once such laws are identified, they genuinely make the design process more systematic, especially on the determination of parameters.

References

1. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.
2. Y. Chen. Generic composition. *Formal Aspects of Computing*, 14(2):108–122, 2002.
3. Y. Chen. Cumulative computing. In *19th Conference on the Mathematical Foundations of Programming Semantics*, ENTCS. Elsevier, 2003.
4. E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programs as fixpoints. In *ICALP'81*, volume 85 of *LNCS*, pages 169–181. Springer-Verlag, 1981.
5. E.C.R. Hehner. Predicative programming I, II. *Communications of ACM*, 27(2):134–151, 1984.
6. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
7. C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
8. L. Lamport. Hybrid systems in TLA+. In *Hybrid Systems*, volume 736 of *LNCS*, pages 77–102. Springer-Verlag, 1993.
9. L. Lamport. A temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
10. A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
11. A. Pnueli and E. Harel. Applications of temporal logic to the specification of real-time systems. In M. Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems, Lecture Notes in Computer Science 331*, pages 84–98. Springer-Verlag, 1988.
12. A.P. Ravn, H. Rischel, and K.M. Hansen. Specifying and verifying requirements of real-time systems. *IEEE Transactions on Software Engineering*, 19(1):41–55, 1993.
13. M. Schenke and E. Olderog. Transformational design of real-time systems part i: From requirements to program specifications. *Acta Informatica*, 36(1):1–65, 1999.
14. H. Shalqvist. Completeness and correspondence in the first and second order semantics for modal logic. In *Proceedings of the third Scandinavian logic symposium*, pages 110–143. North Holland, 1975.
15. B. von Karger. A calculational approach to reactive systems. *Science of Computer Programming*, 37:139–161, 2000.
16. C. Zhou, C. A. R. Hoare, and A. P. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, 1991.
17. C.C. Zhou, A.P. Ravn, and M.R. Hansen. An extended duration calculus for hybrid real-time systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems, Lecture Notes in Computer Science 736*, pages 36–59. Springer-Verlag, 1993.

Integration of Specification Languages Using Viewpoints

Marius C. Bujorianu

Computing Laboratory
University of Kent
Canterbury CT2 7NF, UK
M.C.Bujorianu@kent.ac.uk
Fax 0044 (0)1227 762811

Abstract. This work provides a general integration mechanism for specification languages motivated by partial specification. We use category theory to formalise specification languages and define a relational semantic framework. We show some inherent limits of the approach, and propose a solution inspired from Z semantics to overcome it. An integration of Z and CCS is considered as an example.

Keywords: Language integration, viewpoints, category theory, type theory, Z, process algebra

1 Introduction

Our motivation comes from the practice of partial (also called viewpoint) specification. The development of large software system might benefit when the participants are let free to use their own languages and techniques, according to their own background and, especially, their own perspective on the system to be developed. This approach prompts specific problems like integration of -partial-specification, checking for contradictory requirement and doing this constructively (by generating feedback in case of an inconsistency).

Partial specification practice has shown that specifications are much more easily integrated when they are explained in common terms. Successful techniques have been developed for homogenous (i.e. written in a common notation) partial specifications developed, for example, in Z [5].

More problematic is the case of heterogeneous partial specifications. In some cases it is easy to translate one partial specification into the language of the other and back. This suggests techniques of reducing heterogeneity to homogeneity. Such a technique is, for example, the translation of all partial specifications into a common semantic framework, where integration and consistency can be constructively studied. There have been proposed such frameworks, e.g. the relational one described in [6]. The experience of applying this framework to UML [4] drives to an important case, which is investigated in this paper.

When considering partial specifications described by a particular type of UML diagrams, like an object diagram and a state diagram, we find they can

have orthogonal roles. This situation was treated using languages like Z [18] and B [7], where both data structures and state machines can be described using the same language. But what if these partial specifications had already been described using different formal notations, like an algebraic specification for the data viewpoint and CCS for the state machine viewpoint? In such a case the unification of viewpoints might require an integration of their languages, in our example unification being accurately specified in Lotos.

One issue we address in this work is unification of viewpoints, as those expressed by different UML diagrams, which requires a specific kind of integration of their languages.

Section 3 provides the leading example. Three different perspectives are considered on a telephone system involving a wait-calling phone. The first viewpoint describes in details the data involved - which we call the data viewpoint. The second viewpoint describes the phone's operations using an abstract version of states. We call it the state&operations viewpoint. The third viewpoint describes the phone behavior, using process algebraic operators such as nondeterministic choice, sequential composition and concurrent composition, with actions denoting operations in the second viewpoint. The composition of viewpoints can be accurately done using an integration of languages involved.

We study the categorical basis of such languages integration in the following sections.

The mechanism for combining notations is based on the theory of relations and it is twofold. One way is to add concurrency to a language describing actions. For this, a pointless axiomatization of relations is much more suitable, as it ignores the structure of states. Another way is to combine a notation for describing the state structure with a notation for describing the operations (actions) on the state. This is possible only when a notion of refinement is defined: the operations language must use a simple formulation of states, which is refined by the state language. The last one adds more data constructs, which must be made available to the operations; this requires a syntactic embedding of the data vocabulary into the syntactic description of operations. Requirements must be made then on the models. A refinement of states might imply a refinement of operations. The combining mechanism is quite complex and proving its soundness requires an adequate mathematical foundation, which is investigated in this paper.

Our combining mechanism is made generic by the use of category theory. Types are categorical specifications. A type is therefore populated with elements (models) which form a category. Therefore any formal notation which admits a category of models can be considered for a state language. Examples of states can be sets, algebras, relations or graphs.

An operations language uses a notion of state, which is required to be a category. Operations are defined as relations between objects of this category, i.e. relations between states.

We study the advantages offered by Z in the last section.

The categorical theory underlying our generic approach shows some intrinsic limitations. Once a category equivalent to that of predicate transformers **PT** was used, the combining relational mechanism can't be used anymore. **PT** can be seen as relations over a category of relations over a topos. A possible solution is to use an adjunction between **SET** and **REL** to embed the last category in the underlying topos. Predicate transformers will be lifted then in the relational algebra. This is the mechanism which makes **Z** so simple and efficient. On top of **Z** more relational constructions can be made, like adding concurrency and nondeterminism (process algebra).

Conclusions on our formal experiment are drawn in the end.

2 Category Theory Background

We suppose the reader is familiar with the basic language of category theory. All definitions and notations we use hereafter can be found, for example, in [25].

We note by **SET** the category of set and (total) functions and by **REL** the category of sets and relations.

The category of monotonic predicate transformers [21] **PT** has as class of objects the class of all powersets and its homsets $\mathbf{PT}(\mathbb{P}A, \mathbb{P}B)$ are the sets of total monotonic functions from $\mathbb{P}A$ to $\mathbb{P}B$; the composition is the usual composition of functions.

One way to relate different categories is by using adjunctions. Formally, an adjunction $\mathbf{C} \overset{F}{\rightleftarrows} \mathbf{D}$ between the categories **C** and **D** is a triple $(F, G, (\theta, \phi))$ where $\mathbf{C} \xrightarrow{F} \mathbf{D}$ and $\mathbf{C} \xleftarrow{G} \mathbf{D}$ are functors and (θ, ϕ) are natural isomorphisms between $\mathbf{C}(\_, G\_)$ and $\mathbf{D}(F\_ , \_)$.

In this paper we use the adjunction $\mathbf{SET} \rightleftarrows \mathbf{REL} = (E, \mathbb{P}, (\{\}, \ni))$, where: $E : \mathbf{SET} \rightarrow \mathbf{REL}$ is the embedding functor from sets to relations; $\mathbb{P}$ is the functor which sends every set A to its power set $\mathbb{P}A$ and every relation $R : A \leftrightarrow B$ to its existential image $[R] = \{b \in B \mid \exists a \in A \bullet aRb\}$; $\{\}_A : A \rightarrow \mathbb{P}A$ is the singleton function and $\ni_A : \mathbb{P}A \rightarrow A$ is epsilonoff, the converse of the membership relation. A similar adjunction $\mathbf{REL} \rightleftarrows \mathbf{PT} = (E^b, A^b, (-^{\circ p}, -^{\circ}))$ is described in [21 pg. 114].

Categories of Relations

Relations proved to be a powerful tool in the study of specification languages. A categorical treatment of relations would help to achieve a generic style and will make relational calculus more applicable.

Relations can be formalized categorically in many ways. Despite this great variety of formalizations, including the theory of allegories [2], we use in this work two approaches, most relevant to our investigation.

The point free approach ignores the nature of related objects and focuses on abstract properties of relations. This approach is useful to define process algebraic types, where emphasis is put on the transitions dynamics and less on the nature of states.

The pointwise approach explicitly considers the objects related by the relation. In our work, these objects have all the same type (i.e. they are all models of the same specification) and they are formally objects of a category. Relations are defined then as (some particular shape of) spans over this category. This approach is useful to define state&operations types, where states are models of the state specification (type) and operations are relations between them.

The two approaches must be seen as the two faces of the same coin. The subjects of the axiomatizations are the same, namely relations. Only their mathematical properties are more or less relevant in different contexts.

Some mathematical results should make this claim more precise. The literature abounds in representation results for relations but it is not the ease to consider them in detail. One important observation must be made: relations can not be defined over any arbitrary category and representation results are conditioned by the properties of the base category. In particular, everything works well when the base category is a (quasi) topos, i.e. categories similar to that of sets and functions.

Point Free Axiomatization of Relations

The definition of relational category captures the basic properties of relations, analogous to other formalizations like allegories [2], which are too rich for our purpose. Formally, a category $\mathbf{R}$ is called a *relational category* iff the following conditions are fulfilled :

[*vertical enrichment*] For every pair of objects $A, B \in |\mathbf{R}|$, the homset $\mathbf{R}(A, B)$ is a complete lattice by an ordering relation $\sqsubseteq$. The infimum (glb) and the supremum of two relations $u, v : A \leftrightarrow B$ are noted by $u \sqcap v$ resp. $u \sqcup v$. The least and the greatest relations between A, B always exist and are denoted $0_{A,B}$ resp. $\mathcal{U}_{A,B}$.

[*inverse*] There is an involution operator $^{-1}$ assigning to each relation $u \in \mathbf{R}(A, B)$ a relation $u^{-1} \in \mathbf{R}(B, A)$. We require the following properties to be fulfilled by this operations

(*involution*) $(u^{-1})^{-1} = u$ and $(u; v)^{-1} = v^{-1}; u^{-1}$; u^{-1}

(*monotonicity*) if $u_1 \sqsubseteq u_2$ and $v_1 \sqsubseteq v_2$ then $u_1^{-1} \sqsubseteq u_2^{-1}$ and $u_1; v_1 \sqsubseteq u_2; v_2$

[*span representation*] Every relation $u \in \mathbf{R}(A, B)$ admits a decomposition $u =$

$$\begin{array}{ccc} & C & \\ f^{-1}g \swarrow & & \searrow g \\ A & & B \end{array}$$
 where $f : C \rightarrow A$, $g : C \rightarrow B$ are functions and $f; f^{-1} \sqcap g; g^{-1} = id_C$.

[*terminability*] There is an object, denoted 1 , such that $0_{1,1} \neq id_1 = \mathcal{U}_{1,1}$ and, for any objects $A, B \in |\mathbf{R}|$, $\mathcal{U}_{A,1}; \mathcal{U}_{1,B} = \mathcal{U}_{A,B}$.

In addition, the Dedekind property and the existence of quotient relations are required. We do not use these properties in this work.

Examples of relational categories include $\mathbf{REL}$ and $\mathbf{REL}_{fin}$, the category of finite sets and relations.

A point is a mapping $x : 1 \rightarrow C$.

The concept of relational product will be used to define parallel composition in relational process types, and relational sum will be used to model nondeterminism.

A *relational product* [28] is a triple $(A \times B, \pi, \rho)$ where $A \times B$ is an object and $\pi : A \times B \leftrightarrow A$, $\rho : A \times B \leftrightarrow B$ are relations such that $\pi^{-1}; \pi = id_A$, $\rho^{-1}; \rho = id_B$, $\pi^{-1}; \rho = \mathcal{U}_{A,B}$ and $\pi; \pi^{-1} \sqcap \rho; \rho^{-1} = id_{A \times B}$.

An object $C_1 + C_2$ together with relations $\sigma_1 : A_1 \leftrightarrow C_1 + C_2$, $\sigma_2 : A_2 \leftrightarrow C_1 + C_2$ is a *relational sum* iff $\sigma_j; \sigma_j^{-1} = I_{A_j}$ and $\sigma_i; \sigma_j^{-1} = O_{A_i, A_j}$ for all $j = 1, 2$, and $\sigma_1^{-1}; \sigma_1 \sqcup \sigma_2^{-1}; \sigma_2 = \mathcal{U}_{A_j}$.

A relational category is said to have *relational products* (resp. *relational sums*) iff for every pair of objects the relational product (resp. relational sum) does exist.

The following notations will be useful in the sequel

$$u \odot v = u; \pi^{-1} \sqcap v; \rho^{-1}, \quad u \otimes v = \pi; u \odot \rho; v \quad u \vee v = \sigma_1^{-1}; u \sqcup \sigma_2^{-1}; v, \\ u + v = u; \sigma_1 \vee v; \sigma_2, \quad \text{mix} = (\pi_1 \vee \pi_2) \odot (\rho_1 \vee \rho_2)$$

Pointwise Axiomatisation of Relations

Given a category $\mathbf{C}$, a span $\langle f, g \rangle : B \leftarrow A \rightarrow C$ over $\mathbf{C}$ is formed by three objects $A, B, C \in |\mathbf{C}|$ and two arrows (the “legs”) $A \xrightarrow{f} B$ and $A \xrightarrow{g} C$.

Many categorical formalizations of relations identify them with spans. This is very tempting because of the very simple span algebra. Pullbacks provide the relational composition, swapping around the span’s legs provides the conversion and so on [25]. We note by $\mathbf{SP}_{\mathbf{C}}$, the category of spans over $\mathbf{C}$. But the approach becomes easy very misleading. For example, if the base category $\mathbf{C}$ does not have all products, pullbacks in $\mathbf{SP}_{\mathbf{C}}$ do not exist. Moreover, $\mathbf{SP}_{\mathbf{SET}}$ is isomorphic to the category of multirelations [25]. In order to construct a category of relations given as spans, we have to restrict the spans to a quotient structure, which ‘eliminates’ multiplicities.

Two spans $\langle f_1, g_1 \rangle : B \leftarrow A_1 \rightarrow C$ and $\langle f_2, g_2 \rangle : B \leftarrow A_2 \rightarrow C$ are called *support ordered*, written $\langle f_1, g_1 \rangle \preceq_e \langle f_2, g_2 \rangle$, if there is an epi $e : A_2 \rightarrow A_1$ such that

$$f_2; e = f_1 \quad \text{and} \quad g_2; e = g_1. \quad (\text{Ref})$$

Composition of spans is monotonic with respect to such an ordering, and thus $\preceq_e$ induces a partial order over $\mathbf{SP}_{\mathbf{SET}}$. We denote by $\cong_e$ the symmetric closure of $\preceq_e$, and by $\mathbf{SPP}_{\mathbf{SET}}$ the category of spans modulo the equivalence $\cong_e$. Each $\cong_e$ -equivalence class has a minimal element (unique up to isomorphism) that gives the minimal representation of the underlying relation. The following result (which appears in many different forms in the categorical literature, see for example [25] page 97, shows this construction provides the relations.

Proposition 1. *The categories $\mathbf{SPP}_{\mathbf{SET}}$ and $\mathbf{REL}$ are isomorphic.*

It is natural to ask how $\mathbf{SPP}_{\mathbf{REL}}$ looks? We might think that $\mathbf{SPP}_{\mathbf{REL}}$ is isomorphic with $\mathbf{PT}$, but arrow composition in $\mathbf{SPP}_{\mathbf{REL}}$ is not monotonic.

3 Viewpoints. An Example

Consider the following behaviour of a telephone system. Several users can use a physical phone. Each user has a unique PIN. The identity of a phone is given then by a number and a PIN. A call-waiting phone can be connected to one or two other phones at any particular time. When it is connected to two phones, one of these is on hold, while the other is current. When a call-waiting phone has only one connection, it can accept a new connection. The original connection is then put on hold. The two connections can be swapped. A conversation is implied when the phones are connected to each other and neither is on hold. Current is a relation which is at most a singleton; if empty the call-waiting phone has no connection, otherwise it contains the identity of the connected phone which, in the case of two connections, is not on hold. A disconnection can be instigated only by a phone for which the connection is current. A more sophisticated version of this case study is proposed in [12].

The system is specified by three different viewpoints. Every viewpoint uses its own language, thus the whole process description is very heterogeneous.

We consider the reader is familiar with the languages involved, therefore the viewpoint specifications will be not explained.

The Data Viewpoint

This viewpoint focuses on describing the data structures of the system Data types are specified here using relational algebraic specification as introduced in [20]. We point here the main differences from a standard algebraic specification language.

A *relational signature* [20] is a pair $\Sigma = (S, \mathcal{R})$ where S is a set of sort symbols and $\mathcal{R}$ is a set of relation symbols with given arities (or *types* of relation). A relational structure over $\Sigma = (S, \mathcal{R})$ is a pair $A = (|A|, \mathcal{R}^A)$ where $|A|$ is a S -sorted set (called the carrier) and $\mathcal{R}^A$ is a set of relations, such that, for each $[R_i : s_1 \times \dots \times s_k] \in \mathcal{R}$ there is a heterogeneous relation $R_i^A \subseteq |A|_{s_1} \times \dots \times |A|_{s_k}$. For each $R \subseteq |A|_{s_1} \times \dots \times |A|_{s_k}$ we note $\text{Im } R = \{a \in |A|_{s_k} \mid \exists a_1 \in |A|_{s_1}, \dots, a_{k-1} \in |A|_{s_{k-1}} \bullet (a_1, \dots, a_{k-1}, a) \in R\}$ and, for every set or relation M , we note by $\text{card}(M)$ its cardinality.

Variables, terms, equations and models are introduced as usual in algebraic specification.

Spec *DataState* is

Sorts *PIN, NUMBER*
Relations *Call : PIN × NUMBER*
Vars *current, onhold : Call*
Axioms
 $\text{card}(\text{current}) \leq 1$
 $\text{card}(\text{onhold}) \leq 1$
 $\text{card}(\text{Im } \text{current}) \leq 1$
 $\text{current} \cap \text{onhold} = \emptyset$

The State&Operations Viewpoint

This viewpoint focuses on the description of the main operations of the system It uses a OCL style notation. The state is described by some attributes

and the operations (named methods) are described by their pre and post conditions. We considered the proposal [23] that @pre construction is allowed in postconditions in the new version of OCL.

```

def: OCLType = Phone
def: OCLType : Set of Phone = Ens
def: OCLAttribute : Ens = current, onhold
let to?, from? : Phone
inv: current = empty implies onhold = empty and
      not self IN current -> union(onhold)
method RequestConnection
  pre current = empty
  post current = Set {to?}
method AcceptFirstConnection
  pre current = empty
  post current = Set{from?}
method AcceptSecondConnection
  pre not current = empty and onhold = empty and not from? IN current
  post current = Set{from?} and onhold = current@pre
method SwapConnections
  pre not current = empty and not onhold = empty and not from? IN current
  post current = current@pre and onhold = current@pre
method Disconnect
  pre not current = empty
  post current = onhold@pre and onhold = empty
method OnHoldDisconnect
  pre not onhold = empty
  post onhold = empty

```

The Process Viewpoint

This viewpoint concentrates exclusively on aspects such as order of methods and their composition (like nondeterministic and concurrent). The process algebraic language used is similar to CCS. Only the method's signature is used.

$$\begin{aligned}
 \text{AcceptConnection}(\text{from?}) &\triangleq \text{AcceptFirstConnection}(\text{from?}) \parallel \\
 &\quad \text{AcceptSecondConnection}(\text{from?}) \\
 \text{Connect} &\triangleq \text{RequestConnection}(\text{to?}) \mid \text{AcceptConnection}(\text{from?}) \\
 \forall X, Y : \text{State} \bullet \text{Disconnect}(X, Y) &\triangleq \text{Disconnect}(X) \mid \\
 &\quad (X \in Y.\text{current} \mid \text{Disconnect}(Y)) \parallel \\
 &\quad (Y \in X.\text{onhold} \mid \text{OnHoldDisconnect}(Y))
 \end{aligned}$$

4 A (Relational) Categorical Framework

The framework we propose now generalizes the most used specification languages, like algebraic specifications, process algebra and model oriented languages like Z and B.

4.1 Specification Frames

A specification frame is a categorical formalization of (strongly) typed specification languages. It can be also seen as a generalization of the concepts of institutions and parchments [22], in the sense that sorts in an institution form the particular case of simple types in a specification frame. As the whole theory is very rich [8], we present here its very basic principles.

A *specification frame* is a tuple $\mathcal{S} = (\mathbf{K}, \text{Syn}, \text{Int}, \vartheta, \mathcal{T}_\mathcal{S})$ where

- $\mathbf{K}$ is a category called the *vocabulary* of the specification frame.

- $\text{Syn} : \mathbf{K} \rightarrow \mathbf{SET}$ is a functor giving the *syntax* of specifications. It must have an (left) adjoint $\text{Kind} : \mathbf{SET} \rightarrow \mathbf{K}$, returning the vocabulary of a given specification.

- $\text{Int} : \mathbf{K} \rightarrow \mathbf{CAT}$ is a functor giving the *possible interpretations* of specifications

- $\vartheta : \text{Syn} \times | \text{Int} | \rightsquigarrow \mathbf{U}$ is an extranatural transformation, giving the *evaluation* of a specification in a model, where $\mathbf{U}$ is a category called the *universe of admissible values*.

- $\mathcal{T}_\mathcal{S}$ is a *type structure* over a specification frame $\mathcal{S}$ and is given by: i) a *type algebra* given as a functorial category $\mathbf{TA}$ (this means that types are categories and the type constructors are relators between types); ii) the *types of a vocabulary* given as a functor $\text{Type} : \mathbf{K} \rightarrow \mathbf{TA}$ (this means we allow the specifier to derive its own types or to use generic types); iii) the *typing of specifications* given as a map τ from $\text{Syn}[\mathbf{K}]$ to $\text{Type}[\mathbf{K}]$; iv) the *carrier of a type* given as an extranatural transformation $\varrho : \text{Types} \times | \text{Int} | \rightsquigarrow \mathbf{W}$ with $\mathbf{W}$, the *universe of type members*, is a category such that $\mathbf{U}$ is equivalent [25] with a subcategory of $\mathbf{W}$.

Types are used to build specifications. The semantics of a type is a category. Type constructors are relators [2], i.e. functors preserving the relational operators and properties. A specification sp , which uses a type t , has models in a category $\mathbf{M} = \text{Int}[\text{Kind}[sp]]$. The semantics of t is a category $\mathbf{W}$. We have to relate the categories $\mathbf{W}$ and $\mathbf{M}$. A natural requirement is to ask $\mathbf{M}$ to be isomorphic with a subcategory of $\mathbf{W}$. The theory of relational categories suggests to weaken this to the existence of an adjunction between $\mathbf{M}$ and $\mathbf{W}$.

Examples

Algebraic (or generalized) institutions. Consider $\mathbf{U}$ the category with only two objects $| \mathbf{U} | = \{\text{true}, \text{false}\}$ and unary boolean functions as arrows. The extranaturality condition generates the familiar satisfaction condition for institutions. Institutions provide a universal semantic universe for all algebraic specification languages [22]. An institution for the language used in the data viewpoint of the example is described in [20].

Process algebras. Consider $\mathbf{K}$ to be $\mathbf{SET}$ and $\mathbf{U}$ as before. The kinds are called sets of actions, specifications are called process specifications and interpretations are called processes.

Model oriented languages. A specification frame for Z is constructed in [8]. The syntax and semantics follow the ISO Standard for Z . Given sets form primitive types. The type constructors of $\mathbf{TA}$ include the power set and the product.

Bindings over a set of identifiers V are modelled as the slice category of **SET** over V . An institutions based approach to Z is reported in [3].

OCL. A specification frame for OCL is easy to construct following the standard semantics [23]. OCL model types require a specification frame for UML. This can be obtained by considering a full, monolithic formalisation of UML in Z , as described, for example, in [18].

Morphisms of specification frames can be defined. specification frames form a category, which is cocomplete. A colimit of specification frames is a new specification frame combining expressions from different notations. This provides a simple mechanism for integration of formal specification languages. Examples include Lotos, which can be defined as colimit of the algebraic specification language ACT and CSP. In this paper we are looking for a much deeper sense of integration.

Space limitation stops us in using specification frames in full generality. In the rest of the paper we consider, for a specification frame, a fixed signature (and thus their associated theory of categories and functors will disappear). We will also consider only one (arbitrary) specification. This practically means that we will work with an arbitrary semantic category $\mathbf{M}$ and a categorical theory of types derived from the type structure. Every element in $\mathbf{M}$ is the meaning of a specification, and types form categories. When $\mathbf{M}$ is a relational category, every object of it is the meaning of a partial and nondeterministic specification.

Types as Specification Frames

There are many different formulations of type theory in the literature; we use the version presented in Crole [9].

Most of frameworks based on type theory are designed towards logics representation. As result, their main emphasis is put proof theory. Our aim is to design a framework able to offer support for representing and reasoning about specification languages. As a consequence, the main emphasis is on model theory. This is why a model oriented approach to type theory as in [9] is useful for us.

In Z , types are used in schemas, but also schemas can be considered as types. We generalise this for an arbitrary specification frame, by interpreting types as specification frames. Our slogan is “*Types are specification frames*”. This is possible because of categorical semantics of types. The semantics of a specification frame is a category (of fibres over a category of vocabularies, see [25] for definition of indexed categories) and types inclusion is modelled by reflections. Therefore, a specification frame can be included in another specification frame, playing role of a subtype.

4.2 Relational Process Types

Consider a relational category with relational sums and a fixed object $L \in |\mathbf{R}|$ such that the relational product $L \times ST$ for every object ST of $\mathbf{R}$ exists.

A coalgebra is a function from a state set ST to the powerset of $L \times ST$. Using the characterisation of relations as set valued functions, we can formulate a relational generalization of coalgebras. A *relational coalgebra* over a set of

states ST with actions from a set L is defined as a relation from ST to $L \times ST$. Relational perspectives on coalgebras can be developed in many ways [2, 27]. Here we limit ourselves to constructing basic composition operators of process algebra on top of a category of states and study what properties the category of states must satisfy to ensure the existence of these constructions.

In the category $\mathbf{RCO}_{ST,L}$ of relational coalgebras the objects are relational coalgebras with an initial state over the state space ST (i.e. pairs (G, r) where $G : ST \leftrightarrow L \times ST$ is a relation and $r : 1 \rightarrow ST$ is a point) and the arrows between objects $G_1 : ST_1 \leftrightarrow L_1 \times ST_1$ and $G_2 : ST_2 \leftrightarrow L_2 \times ST_2$ are mappings between relational coalgebras preserving the dynamics, i.e. arrows $f : ST_1 \rightarrow ST_2$ such that $G_1; (I_L \times f) \sqsubseteq f; G_2$ and $r_1; f = r_2$.

An *atomic action* a is defined as a point $a : 1 \rightarrow ST$.

A *transition* $G_1 \xrightarrow{l} G_2$ is an injective morphism f (in $\mathbf{R}$) such that $G_1; (I_L \times f) = f; G_2$ and $r_1; f \sqsubseteq r_2; G_2$; l ; ρ .

A *unary process operator* is defined as an endofunctor acting on $\mathbf{RCO}_{ST,L}$. A *binary process operator* is a bifunctor acting on $\mathbf{RCO}_{ST,L}$.

The *prefixing operator* Pref_a , by an action a , is defined by

$\text{Pref}_a[G] = [(a, r)]; (id_L \times \sigma_2)$ (thus $\text{Pref}_a[G]$ has the type $1 + ST \leftrightarrow L \times (1 + ST)$)

$\text{Pref}_a[r] = \sigma_1 : 1 \rightarrow 1 + ST$

$\text{Pref}_a[f] = id_1 + f$

Proposition 2. $G \xrightarrow{a} \text{Pref}_a[G]$

The *concurrent composition* operator is defined on relational categories with relational product. We define the binary process operator Co by

$Co[G_1, G_2] = (\pi; G_1; (id_L \times \pi^{-1}) \sqcap \rho; (\rho^{-1})^2) \sqcup (\pi; \pi^{-1}; \rho^{-1} \sqcap \rho; G_2; (id_L \times \rho^{-1}))$

$Co[r_1, r_2] = r_1 \odot r_2$

$Co[f_1, f_2] = f_1 \times f_2$

Proposition 3. If $G_1 \xrightarrow{a} G_2$ then $Co[G, G_1] \xrightarrow{a} Co[G, G_2]$

The *Choice operator*. In many process algebra semantics labelled trees are used instead of graphs. The fold operation eliminates the arcs pointing towards the root of a relational coalgebra. Formally, the *fold* graph $\text{fold}(G)$ of a relational coalgebra $G = (G, r)$ is defined as $\text{fold}(G) = (r; G \vee G); (id_L \times \sigma_2)$ and $\text{fold}(r) = \sigma_1$.

We define now the process operator *Choice* by

$\text{Choice}[G_1, G_2] = \delta; (\text{fold}(G_1) + \text{fold}(G_2)); \text{mix}; (id_L \times \delta^{-1})$

$\text{Choice}[r_1, r_2] = (\delta; (\text{fold}(G_1)^{-1} + \text{fold}(G_2)^{-1}))^{-1}$

$\text{Choice}[f_1, f_2] = \delta; (id_1 + f_1 + f_2); \delta^{-1}$

where δ factorize the relation

$id_{1+ST_1} \sqcup id_{1+ST_2} \sqcup \bigsqcup_{i=1,2, j=1,2} (\sigma_i^{-1}; \text{fold}(G_i)^{-1}; \text{fold}(G_j); \sigma_j)$

Proposition 4. If $G_1 \xrightarrow{a} H_1$ and $G_2 \xrightarrow{a} H_2$ then $G_1 \xrightarrow{a} \text{Choice}[H_1, H_2]$ and $G_2 \xrightarrow{a} \text{Choice}[H_1, H_2]$.

4.3 State and Operations (SO) Types

SO types are the most general forms of specifying state machines. There are three main aspects concerning SO types: states are structured, and their specification might be complex; operations (or transitions) produce a change of the state of the system. The modification of state might also have a complex description; a mechanism of assembly altogether the states and operations description. This can take the form of some compatibility conditions, or just a banal ‘put together’ operation on specifications.

SO types definition is very flexible, therefore many formalisms can be invented. The SO types of interest for us are those arising from Z, LOTOS and algebraic specifications. Being very expressive notations, a common generalization might be too abstract to be useful. We first show how their definition can match a common pattern.

State Types

State types are simply specification frames. They describe the states of a complex system. The most relevant examples come from algebraic specifications and model oriented languages.

Relational SO Types

Operation types are modelled by restricted forms of specification frames. The most common restriction is applied on vocabulary morphisms: very often we require them to be injective.

The main characteristic of operation types is that they are built on top of a state type. An operation (i.e. a member of a operation type) must have access to the state components, thus it must import the whole specification of the state.

Using state items is not enough to describe the effects of an operation: operations often require representation of different states. This is possible in our approach thanks to the concept of type: states are not only models of a specification, but also members of a particular type, thus we can define generic states using well typed variables. The most frequent situation is when we want to refer to the state before an operation is invoked (operation’s pre state) and the state after operation returns (the post state). There is no unitary convention about this in specification languages: in Z the post states is indicated using a prime decoration of the pre state variable, in VDM pre state components are decorated with a $\leftarrow$ symbol and in OCL using the postfix @pre.

A SO type member has a two component signature: the state part signature (imported from the state type) and a operation specific signature (containing, for example, input and output variables, plus auxiliary variables necessary to describe state transformation). Consequently, a model must conform to signature components: the reduction of a SO model to the state sub-signature must be a state model.

Can state and operations types be thought of in separation? Clearly, the concept of state is essential for defining an operation type. We may consider the case when the state type has the simplest possible structure: it is a (unstructured) set of states. This case could be used to define an operation type, and further a SO type as a combination of state and operation type, where the set of states

of the operation type members are refined (populated) into (with) models of the state type. This approach has two sensible points: i) as the semantics of Z show, a set may have structured elements (they can be, for example, pairs) and thus, a finer concept of unstructured type must be defined and ii) the way state elements are replaced with models of a specification must be carefully chosen: we must guarantee the access of operations to the state signature components.

Our approach is based on the genericity idea and uses category theory: A SO type is always constructed in top of a state type. The semantics of a state type members is a category ST . The semantics of SO type members is a category OP_{ST} built on top of ST .

For relational SO types, the semantic category is defined to be S_{ST} . Thus, the semantics of an operation is an ‘abstract relation’ between two objects in ST , i.e. between two concrete states. An operation type is then defined as an SO type for which states form a discrete category (i.e. the states can’t be related in any way).

In order to construct, on top of a relational category R , a new relational category, the span construction from Section 2 is not enough. The reason is that the relational composition fails to be monotonic. This suggests that, in the span definition, we should consider additional information from the arrow structure (in categorical terms, the double category structure). This can be done by replacing the equality in formulas (Ref) by the inequality $\sqsubseteq$. In the new circumstances, the surjectivity requirement on arrow e (which we call now a *ref-morphism*) becomes superfluous and it will be dropped.

The relevance of the above construction comes from the perspective of refinement. In the relational semantics for Z proposed in [10], the order relation $\sqsubseteq$ models refinement. This is why, when naming constructs involving $\sqsubseteq$, we use the prefix *ref* (for refinement).

We can now modify the notion of pullback using refinement order. We take our inspiration from the construction of *subequalizers* of J. Lambek in [19]. The *ref-pullback* of a pair of relations $b : B \leftrightarrow D$ and $c : C \leftrightarrow D$ in R is a span $\langle f, g \rangle : B \leftarrow A \rightarrow C$ such that for all spans $\langle f', g' \rangle : B \leftarrow A \rightarrow C$ we have $g' ; c \sqsubseteq f' ; b \Leftrightarrow (f', g') \preceq_e (f, g)$.

Ref-pullbacks provide a monotonic composition operator on spans, which can be used to define a new category of spans. Iterating the construction from Section 2, we get a new category S_R . The new construction of relations is fully compatible with the previous one, as $S_{SET} = SPP_{SET}$ if we take $\sqsubseteq$ to be the equality $=$. Further, using the categorical machinery from [21], we can show that $S_R = PT$ when $R = SPP_{SET}$. A surprising fact is that PT does not admit ref-pullbacks [16]. This means that construction of relational categories can not continue further and the categorical construction of relations should go further to more abstract formulation.

5 Types Integration

Inspired from UML [24], we propose a hierarchical mechanism using relational semantics. In simplified terms, the mechanism is the following. A specification language, which we call the state language, has given semantics in a category. Another specification language, called operations language, is defined in terms of the state language. A specification in the operations language defines a relation between models of specifications in the state language. Therefore the semantics of an operation specification can be defined only in terms of the state language semantics. The newly resulted language is called state&operations (SO) language. We define a category of relations (the semantics of operations specs) built on top of another category (corresponding to the semantics of state specs). The existence of such constructions imposes some requirements for the state category.

The reader should have noticed us having avoided the classification of static versus dynamic frameworks, as proposed, for example, by the D-oids [1] and algebra transformation systems approaches [22]. Any SO language, with a suitable categorical semantics, can be used as a state language to form a new SO language. This means, as in the case of state diagrams, that states can be distributed and can have their own local dynamics.

C. Fischer [14] distinguishes two main ways of integrating Z with a process algebra: syntactic and semantic. A semantical combination usually combines the two languages using a common semantics. J. Derrick and E. Boiten [11] use failure/divergence semantics, and other approaches take some part of a Z specification and identify it with a process (as described in [14]). The syntactical approach combines the syntax of Z and process algebra and then defines the semantics of the new language by a lifting of two semantics. Prominent syntactical approaches are combinations of Z with CCS as in [26] and [15].

We generalize the above approaches to arbitrary specification languages, provided their categorical semantics satisfy some compatibility conditions. Roughly, Fischer's semantic integration corresponds to $\mathbf{D} + (\mathbf{SO} + \mathbf{P})$ whilst the syntactic one corresponds to $(\mathbf{D} + \mathbf{SO}) + \mathbf{P}$.

The integration $\mathbf{SO} + \mathbf{P}$ means that process algebra operators are built in top of operations. Atomic process are operations names, and their semantic is relational (in pointless form). The integration $\mathbf{D} + (\mathbf{SO} + \mathbf{P})$ is very loose: a basic SO specification, described in the main viewpoint, is refined by two subordinated viewpoints. States are refined to more concrete data structure by a data viewpoint, and operations are refined towards a complex behavior by a process viewpoint.

for the *transitional SO types* the considered semantic category is $\mathbf{RCO}_{ST}$, the category of coalgebras having states any object of $\mathbf{ST}$.

The integration $(\mathbf{D} + \mathbf{SO}) + \mathbf{P}$ is much deeper. The process viewpoint has access not only to operations, but also to the refined data. Process communication manipulates now the complex data structures defined by the data viewpoint. The relational semantics of processes must consider the semantics of data they use: the relational axiomatisation must be pointwise. This is the key point of this

approach: the semantics of $\mathbf{D} + \mathbf{SO}$ comes in the form of a category; on top of this category the relational semantics of processes is constructed. The essence of the categorical approach is the characterization of properties using morphisms. The morphisms characterizing state properties are the span legs defining the relational semantics of processes.

Syntactic restrictions must also be imposed. The $\mathbf{D} + \mathbf{SO}$ must be explicitly imported into the process specifications. A signature of a $(\mathbf{D} + \mathbf{SO}) + \mathbf{P}$ specification contains a $\mathbf{D} + \mathbf{SO}$ subsignature. This must be reflected in the semantics. A $(\mathbf{D} + \mathbf{SO}) + \mathbf{P}$ specification model, when restricted to $\mathbf{D} + \mathbf{SO}$ subsignature, must be a $\mathbf{D} + \mathbf{SO}$ model.

Because the semantics of relational process types (RPTs) was given in a formalization of relations which does not consider the categorical structure of states and transitions, these can be integrated in a very liberal manner with almost any specification language. The only requirements for coupling a RPT with a specification language formalized as a specification frame is that the category of models of the former must be small. Moreover, the set of actions must be provided. This can be done using a specification frame for process languages, as described in section 4.1, or the set of operations of a SO type.

In the example from Section 3, a process algebraic viewpoint can be integrated with the SO viewpoint as

$AcceptConnection \triangleq AcceptFirstConnection \parallel AcceptSecondConnection$
 $Connect \triangleq RequestConnection \mid AcceptConnection$
 $P = AcceptConnection; Connect; RequestConnection; Disconnect;$
 $SwapConnection; Connect; P$

We observe that process expressions do not contain any variables. This is because of semantical integration. The operations semantics and process semantics are given, in this case, in the same universe of relations. The same relations are seen (and formalized) differently by the two viewpoints.

In order to make use of variables in process expressions we need a syntactic integration. In our example, such a category does not exist. We look for a solution taking our inspiration from Z.

The semantics of processes will be given in a model relations category described in the reminder of this section (a categorical development of the relational theory exposed in the section 4.3).

Let us consider the pullback $A \leftarrow R \rightarrow B$ of $A \xrightarrow{f} O \xleftarrow{g} B$ in $\mathbf{SET}$. The object R is characterized as $R = A \times_0 B = \{(a, b) \in A \times B \bullet f(a) = g(b)\}$, thus, R is a relation of type $A \leftrightarrow B$. This construction suggests that relations axiomatized as spans can be defined as pullbacks in the base category (if exist). In a more general setting, let us denote a span category as $\mathbf{V} = 1 \xleftarrow{f} 0 \xrightarrow{g} 2$ (with identity arrows omitted for simplicity), $\mathbf{C}$ a category and $F : \mathbf{V}^{op} \rightarrow \mathbf{C}$ a functor. This

is illustrated graphically by the diagram

$$\begin{array}{ccccc}
 & & R & & \\
 \pi_1 \swarrow & & & & \searrow \pi_2 \\
 F(1) & & & & F(2) \\
 & \searrow F(f) & & \swarrow F(g) & \\
 & & F(0) & &
 \end{array}$$

The

pullback R of F in **SET** [25] is characterized as $R = \{(a, b) \in F(1) \times F(2) \bullet F[f](a) = F[g](b)\}$. This formula inspires the following definition.

The category $\mathbf{MR}_{\mathcal{S}}$ of model relations between state specifications over a fixed vocabulary V of a specification frame $\mathcal{S}$ has as objects relations $(\mathbf{D}, F, \mathbf{M})$ where the category $\mathbf{D}$ is the shape of the relation, $F : \mathbf{D} \rightarrow \text{Syn}^{\mathcal{S}}[V]$ is a functor and $\mathbf{M}$, a full subcategory of $\text{lim}(F; \text{Int}_V^{\mathcal{S}})$.

A morphism $mr f = (G, \alpha)$ in $\mathbf{MR}_{\mathcal{S}}$ from $(\mathbf{D}_1, F_1, \mathbf{M}_1)$ to $(\mathbf{D}_2, F_2, \mathbf{M}_2)$ has two components; the first component is a functor $G : D_1 \rightarrow D_2$ and the second component is a natural transformation $\alpha : F_1 \Rightarrow G; F_2$.

One can remark immediate advantages of this definition. First, its construction is less restrictive than the previous definitions. Second, we can model relations with arbitrary arities. If one considers the shape

$$\begin{array}{ccccc} In & Out & & St & St \end{array}$$

$\begin{array}{ccccc} & In & Out & & St & St \\ & \swarrow & \swarrow & & \nearrow & \nearrow \\ & & R & & & \end{array}$
 for $\mathbf{D}$, then models are relations between states

Sts, input *In* and output *Out*. The big disadvantage is that composition cannot be defined anymore, and an abstraction mechanism should be imposed.

The category $\mathbf{MR}_{\mathcal{S}}$ is the key concept for the construction of a specification frame $\mathcal{SO}_{\mathcal{S}}$ for SO types in which $\mathcal{S}$ describes the states. The full construction will appear in [8] and it is very technical, based essentially on the use of fibred categories. In order to make interpretations of $\mathcal{SO}_{\mathcal{S}}$ in $\mathbf{MR}_{\mathcal{S}}$, we need to define $\lim(F; Int_{\mathcal{S}}^{\mathcal{V}})$ as expressions in $Syn^{\mathcal{SO}_{\mathcal{S}}}$. This can be done using the following idea. As the functors $Syn^{\mathcal{SO}_{\mathcal{S}}}$ and $Int^{\mathcal{SO}_{\mathcal{S}}}$ are contravariant, the pullback $\lim(F; Int_{\mathcal{S}}^{\mathcal{V}})$ is described by a pushout $colim(F; Syn_{\mathcal{S}}^{\mathcal{V}})$. This correspond to inclusion of two versions of the state type. Using the Z decoration conventions, let us denote them by $\mathcal{S}$ and $\mathcal{S}'$. Any member St of $\mathcal{S}$ will be interpreted as operation's pre-state, and any member St' of $\mathcal{S}'$ will be interpreted as operation's post-state. A set of well formed expressions built over $colim(Kind\{St\}; Kind\{St'\})$ will define a full subcategory of $\lim(F; Int^{\mathcal{S}})$.

We will apply this construction to our example by using Z in the next section.

6 Using Z

In this section we use the state and operations semantics of Z , as described in [10] and [5]. The SO type is defined as follows. State description is made using the usual Z schemas. Their semantics is given in **SET**, considering the image of the forgetful functor from the category of bindings. Operation schemas are built on top of a state schema, using a convention of decorations. Their semantics is given as relations between models of states (that are sets of bindings), i.e. $\mathbf{SPP}_{\mathbf{SET}}$ which is isomorphic to **REL**.

We can construct now **S_{REL}**, which corresponds to model relations built over Z's SO types. This category provides then the semantical universe for a semantical integration of Z and process algebra. Existing work on integration of Z with CCS [15] and [26] offers enough intellectual comfort to practically exemplify our theoretical integration mechanism.

We owe an explanation of the semantical integration of process algebraic viewpoint being possible in Z and not in the formalization from Section 3.

A subtle difference between the state and operations semantics and the semantics from Z Standard [10] consists in the implicit use of the adjunction **SET** $\rightleftarrows$ **REL** described in Section 2. As a consequence, the semantics of operation schemas is given in **REL** and not in **PT**. This is essential, for example, for the theory of refinement [10].

Compared with relational algebraic specification [20] and relational datatypes specification language [2], Z offers much more formal support for specification and refinement. We exemplify this in the remainder of this section, by a full specification of the example from Section 3.

Data and State Viewpoints

We use the rich relational specification tools provided by Z to refine further the data viewpoint from section 3. The advantages of using Z come from a more expressive notation and a simpler semantics (states are properly treated as sets and as relations).

$SmallDigit == ' \# ' \mid ' 0 ' \mid ' 1 ' \mid ' 2 ' \mid ' 3 ' \mid ' 4 ' \mid ' 5 '$

$Digit == ' 0 ' \mid ' 1 ' \mid ' 2 ' \mid ' 3 ' \mid ' 4 ' \mid ' 5 ' \mid ' 6 ' \mid ' 7 ' \mid ' 8 ' \mid ' 9 '$

$PIN = \{s : seq_1 SmallDigit \mid head\ s = ' \# ' \wedge \# tail\ s = 4\}$

$NUMBER = \{s : seq_1 Digit \mid head\ s = ' 8 ' \wedge head\ tail\ s = ' 2 ' \wedge \# tail\ s = 4\}$

$Call = PIN \longleftrightarrow NUMBER$

<i>DataState</i>
<i>current</i> : <i>Call</i>
<i>onhold</i> : <i>Call</i>
$\#current \leq 1$
$\#onhold \leq 1$
$\#ran\ current \leq 1$
$current \cap onhold = \emptyset$

Operations Viewpoint. The Z schemas specifying the operations simply translate the conjunction of pre and post conditions of the corresponding OCL methods. The main advantages come from the elegance of the mathematical language and from much developed semantics and refinement.

[*Phone*]

<i>State</i>	<i>Init</i>
<i>current</i> : $\mathbb{P}\ Call$	<i>State'</i>
<i>onhold</i> : $\mathbb{P}\ Call$	$current' = \emptyset$
<i>self</i> : <i>Phone</i>	
$current = \emptyset \Rightarrow onhold = \emptyset$	
$self \notin current \cup onhold$	

<i>RequestConnection</i> _____ $\Delta State$ $to? : Phone$ $current = \emptyset$ $current' = \{to?\}$	<i>AcceptFirstConnection</i> _____ $\Delta State$ $from? : Phone$ $current = \emptyset$ $current' = \{from?\}$
<i>AcceptSecondConnection</i> _____ $\Delta State$ $from? : Phone$ $current \neq \emptyset$ $current' = \{from?\}$ $onhold = \emptyset$ $onhold' = current$ $from? \notin current$	<i>OnHoldDisconnect</i> _____ $\Delta State$ $onhold = \emptyset$ $onhold' = current$
<i>SwapConnection</i> _____ $\Delta State$ $current \neq \emptyset$ $current' = onhold$ $onhold = \emptyset$ $onhold' = current$	<i>Disconnect</i> _____ $\Delta State$ $current \neq \emptyset$ $current' = onhold$ $onhold' = \emptyset$

Unification of viewpoints as described in this paper, results in the semantics of Standard Z. In addition, we exemplify the “orthogonal” mechanism of viewpoint integration described in [5].

The correspondence between data and state viewpoints is $Phone = PIN \times NUMBER$ plus the implicit correspondences resulted from identical names. Thus $Call = \mathbb{P} Phone$.

The unified state schema is the pushout of the the specifications $DataState$ and $State$, in the relational category of Z schemas with data refinement as arrows.

<i>DataState</i> _____ $current : Call$ $onhold : Call$ $self : Phone$ $\#current \leq 1$ $\#onhold \leq 1$ $\#ran\ current \leq 1$ $current \cap onhold = \emptyset$ $current = \emptyset \Rightarrow onhold = \emptyset$ $self \notin current \cup onhold$

The operations must be adapted to the new state space and the adapted operations checked for consistency - an easy task in our example.

The correspondence between state&operations and process viewpoints results from names identity.

The process viewpoint must be described in an integration of Z with CCS [15]. What makes the difference with specification from section 3 is a very clear semantics.

7 Conclusions

In this work we proposed a categorical formalization of a relational framework [6] for partial specification. The framework used successfully Z as a basis for studying homogenous and heterogeneous viewpoints. Our formalization of these techniques takes the form of relational SO types.

Inspired from our work on UML [4], we propose a new way of composing heterogeneous viewpoints. This is a form of hierarchical composition, where an operations viewpoint is integrated on top of a state viewpoint, a data viewpoint is integrated in the state viewpoint. Further, a process viewpoint might be added. All these integration can be different in nature. Adding data results in a refinement. Adding operations results in a construction of state machines, whose behavior is described using relations. Adding explicit concurrency and non-determinism (processes) can be done in different ways. We study two of them, inspired from [14]. One way, called semantic integration, is based on the idea of using a common semantics for processes and operations (in our case, a relational one). A more profound way, called syntactic integration, adds concurrency to state machines. Concurrent processes have full access to the states and operations structure. We propose a relational categorical semantics for them in the form of model categories (at the end of section 4. 3) Worked examples of such integration include languages as ZCCS [15] and CCZ [26], but the all possible examples are not restricted to these. The use of category theory offers to our integration method many applicability domains.

We give an example taken from telecom systems. A heterogeneous specification of the system is given, using relational algebraic specification for data and state viewpoints, a pre&post conditions language à la OCL for operations viewpoint and a simplified CCS for the process algebra. We show how these viewpoints can be integrated in our framework, and the first result is that the process viewpoint can be integrated only in a semantic manner. We show that a syntactic integration is possible when using Z with state and operations interpretation. We formulate the difference between approaches categorically using adjunctions. The example written in Z shows the richness, beauty and simplicity of a relational language.

Although very simple and elegant, the categorical formalisation of relations as spans imposes constraints on the base category which, very often, are not fulfilled by the semantics of specification languages. In our example, the semantics of the operations viewpoint can not be defined in a relational manner. We

have proposed a solution by generalising the construction of relations using re-pullbacks in section 3.3. But the new construction can not be iterated further, thus we can not define relational semantics for the process viewpoint. We proposed several solutions. One was to construct relational types in section 3.2, corresponding to a semantic integration. One solution was exemplified in section 6, namely to use one the adjunctions **SET** $\rightleftarrows$ **REL** and **REL** $\rightleftarrows$ **PT**. A last solution, the most general, was to introduce model relations in section 5.

Much further work remains to be done. We have expressed our ideas using a drastic simplification of specification frames, the categorical formalization of specification languages. The next step is to formulate the relational categorical semantics to the generality required by specification frames. Because of space limitation we introduced SO types rather in an informative way. An extension of the theory to consider specification frames was sketched in section 5. A complete formalization of SO types as specification frames will be done in a subsequent paper. The practical impact on specification language integration of the results of this theoretical investigation must be assessed. In particular, the interaction of components in a concurrent composition must be further refined. We feel that more development ideas would be derived from a comparison of our approach with other categorical integration techniques as D-oids [1] or non-categorical, as C.A.R. Hoare's Unified Theory of Programming [17].

Mainly, our formal experiment shows that category theory is a useful vehicle for investigating the mathematical basis of specification language integration.

Acknowledgements. This work wouldn't be possible without the constant support of Dr. Eerke Boiten. The constructive suggestions of the anonymous referees are gratefully acknowledged.

References

1. E. Astesiano, E. Zucca, D-oids: a model for dynamic data types, *Mathematical Structures in Computer Science*, vol.5, pp. 257-282, 1995.
2. R. Backhouse and P. Hoogendijk. Elements of a Relational Theory of Datatypes. In B. Möller, H.A. Parts, and S.A. Schuman, editors, *Formal Program Development. Proc. IFIP TC2/WG 2.1 State of the Art Seminar*, LNCS 755, pp. 7-42, 1993.
3. H. Baumeister. Relating Abstract Datatypes and Z-Schemata. WADT'99 Bonas, France, Nov 1999.
4. E.A. Boiten and M.C. Bujorianu Exploring UML Refinement through Unification Workshop on Critical Systems Development with UML, <<U M L>>2003, San Francisco, California, USA, October 20 - 24, 2003.
5. E.A. Boiten, J. Derrick, H. Bowman, M.W.A. Steen. Constructive consistency checking for partial specification in Z. *Science of Computer Programming*, 35(1):29-75, 1999.
6. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency. *Formal Methods in System Design*, 21:111-166, 2002.
7. M. Butler and C. Snook, Verifying Dynamic Properties of UML Models by Translation to the B Language and Toolkit. In *Proceedings UML 2000 WORKSHOP Dynamic Behaviour in UML Models: Semantic Questions*, York, October 2000.

8. M.C. Bujorianu *A Categorical Framework for Partial Specification*, forthcoming PhD Thesis, Computing Laboratory, University of Kent, 2004.
9. R. Crole *Categories for Types* Cambridge University Press, 1993.
10. J. Derrick and Eerke Boiten. *Refinement in Z and Object-Z: Foundations and Advanced Applications*. Formal Approaches to Computing and Information Technology. Springer, May 2001.
11. J Derrick and E Boiten Combining Component specifications in Object-Z and CSP *Formal Aspects of Computing*, 13:111-127, May 2002.
12. R. Duke and G.A. Rose *Formal Object-Oriented Specification Using Object-Z* Cornerstones of Computing Macmillan 2000.
13. J.L.Fiadeiro and J.F.Costa, Mirror, Mirror in my Hand: a duality between specifications and models of process behaviour, in *Mathematical Structures in Computer Science* 6, pp. 353-373, 1996.
14. C. Fischer: How to Combine Z with a Process Algebra in *ZUM '98: The Z Formal Specification Notation*, LNCS 1493, Springer Verlag, 1998.
15. A.J. Galloway and W. Stoddart: An operational semantics for ZCCS in *International Conference on Formal Engineering Methods (ICFEM)*, IEEE Computer Society Press, 1997.
16. P. H. B. Gardiner, C. E. Martin, O.de Moor: An Algebraic Construction of Predicate Transformers. *Science of Computer Programming* 22(1-2) pp. 21-44, 1994.
17. C.A.R. Hoare: Unifying theories: a personal statement, *ACM Computing Surveys* 28A(4) 1996.
18. S-K Kim and D. Carrington: A formal mapping between UML models and Object Z specifications In *Proceedings ZB 200: Formal Specification and Development in Z and B* LNCS 1878, 2000.
19. J. Lambeck Subequalizers *Bull. of American Mathematical Society*, 13 (3) : 337-349, 1970.
20. Y. Lamo *The institution of multialgebras - a general framework for algebraic software development* PhD thesis, University of Bergen, 2003.
21. O. de Moor: Inductive Data Types for Predicate Transformers. *IPL* 43(3): 113-117, 1992.
22. T. Mossakowski, A. Tarlecki, Pawlowski. *Combining and representing logical systems using model-theoretic parchments*. Recent Trends in Algebraic Development Techniques, LNCS 1376, p. 349-364, Springer-Verlag, 1998
23. OMG's Object Constraint Language (OCL) 2.0 RFI Response Draft (University of Kent, Microsoft e.a.) 2002.
24. J.Rumbaugh and I.Jacobson, *The Unified Modeling Language Reference Manual*, Addison Wesley Longman Inc., 1999.
25. D.E. Rydeheard and R.M. Burstall *Computational Category Theory* Prentice Hall, 1988.
26. K. Taguchi, K. Araki, The State-based CCS Semantics for Concurrent Z Specification in *Procs. of International Conference on Formal Engineering Methods (ICFEM)*, IEEE Computer Society Press, pp. 283-292, 1997.
27. Winter M.: Generating Processes from Specifications using the Relation Manipulation System RELVIEW. *ENTCS* 44 (3), 2003.
28. H. Zierer Relation Algebraic Domain Constructions *Theoretical Computer Science* Vol. 87, pages 163-188, 1991.

Integrating Formal Methods by Unifying Abstractions

Raymond Boute

INTEC, Ghent University, Belgium,
boute@intec.UGent.be

Abstract. Integrating formal methods enhances their power as an intellectual tool in modelling and design. This holds regardless of automation, but a fortiori if software tools are conceived in an integrated framework. Among the many approaches to integration, most valuable are those with the widest potential impact and least obsolescence or dependency on technology or particular tool-oriented paradigms. From a practical view, integration by unifying models leads to more uniform, wider-spectrum, yet simpler language design in automated tools for formal methods.

Hence this paper shows abstractions that cut across levels and boundaries between disciplines, help unifying the growing diversity of aspects now covered by separate formal methods and mathematical models, and even bridge the gap between “continuous” and “discrete” systems. The abstractions also yield conceptual simplification by hiding non-essential differences, avoiding repeating the same theory in different guises.

The underlying framework, not being the main topic, is outlined quite tersely, but enough for showing the preferred formalism to express and reason about the abstract paradigms of interest. Three such paradigms are presented in sufficient detail to appreciate the surprisingly wide scope of the obtained unification. The function extension paradigm is useful from signal processing to functional predicate calculus. The function tolerance paradigm spans the spectrum from analog filters to record types, relational databases and XML semantics. The coordinate space paradigm covers modelling issues ranging from transmission lines to formal semantics, stochastic processes and temporal calculi. One conclusion is that integrated formal methods are best served by calculational tools.

Keywords. Abstraction, continuous, discrete, hybrid systems, databases, filters, formal methods, function extension, function tolerance, integration, predicate calculus, quantification, semantics, signal processing, transmission lines, unification, XML.

1 Introduction: Motivation, Choice of Topic, and Overview

Applying formal methods to complex systems may involve modelling different aspects and views that are often expressed by different paradigms. Insofar as

complexity is due mainly to quantitative elements (e.g., size of the state space), past decades have seen impressive progress in the capabilities of automated tools, where we single out model checking as a representative example [20,32,45].

However, when complexity is due to the need for combining different modelling viewpoints [3,28,44], tools are too domain-specific, each reflecting a particular paradigm in a way not well-suited towards conceptual combination. Moreover, automation always carries a certain risk of entrenching ad hoc paradigms and poor conceptualizations, the equivalent of legacy software in programming. Both the tool developer and the user tend to preserve the invested effort, and successes with near-term design solutions curtail incentive for innovation. Hence much existing tool support fails to exploit the advantages that formality brings.

Integration is not in the first place a matter of tools, but of careful thinking about concepts, abstractions and formalisms before starting to think about tools.

From an engineering perspective, there is an instructive analogy with automated tools in classical engineering disciplines, such as mechanics and electronics. These disciplines are mainly based on physical phenomena, which are best modelled by methods from (linear) algebra and analysis or calculus. The use of well-known automated tools such as Maple, Mathematica, MATLAB, Mathcad (and more specialized ones such as SPICE) is very widespread, comparatively much more so than the use of tools for formal methods in software engineering. We attribute this to two major factors, namely

- a. The abstractions: for modelling physical phenomena, algebra and analysis have tremendous power of abstraction. For instance, one differential equation can model vastly different phenomena, yielding effortless integration.
- b. The formalisms¹: the notation and rules supported by software tools are those that have proven convenient for human communication and for pencil-and-paper calculations that are essentially *formal*². Still, this only refers to calculating with derivatives and integrals; the logical arguments in analysis are quite informal, causing a severe style breach (addressed below).

So tool design *follows* well-designed abstractions and formalisms. Wide-scope abstractions, formalisms that are convenient for formal calculation by hand (not just when automated) and style continuity are hallmarks of mature integration.

In computing (hardware, software), formal logic has always been the basis for mathematical modelling, and is now supported by good formal tools [40,42]. Although these tools conceptually have a wider scope than, say, model checking, they do not play the same role as those mentioned for classical engineering.

- a. The abstractions: the level where the generality of formal logic is exercised differs from that of algebra and analysis. In fact, there is a strong case for using formal logic in analysis to eliminate the style breach, which is now made possible in an attractive way by advances in calculational logic. However, this

¹ A *formalism* is a language/notation together with rules for symbolic manipulation.

² This means manipulating expressions on the basis of their *form*, using precise rules, unlike the common way based on *meaning* (intuitive interpretation). In this way, the shape of the expressions provides guidance in calculations and proofs.

only shows that logic by itself does not constitute the desired wide-spectrum paradigm, but needs to be complemented by other mathematical concepts.

- b. The formalisms: logics supported by current tools are by no means convenient for pencil-and-paper calculation or human communication³. Here also is a severe style breach: the mathematics used in studying principles and algorithms for the tools themselves is highly informal (e.g., the use of quantifiers and set comprehension reflects all the deficiencies outlined in section 2) and the proofs in even the best treatments are mere plausibility arguments.

Here the tools *impose* the abstractions and formalisms, often quite narrow ones.

As an aside: Lamport [35] correctly observes that, for systems specification, mathematics is more appropriate than program-like notation. The latter misses the power of declarativity (necessary for abstraction and going beyond discrete processes) and convenience for pencil-and-paper calculation (for which it is too verbose). Integrating formal methods and tools makes similar demands.

The theme description for this conference explicitly mentions the following approaches to integrating different viewpoints: creating hybrid notations, extending existing notations, translating between notations, incorporating a wider perspective by innovative use of existing notation.

Of course, these are not mutually exclusive. The approach used here contains some flavour of all, but most emphatically the latter. Referring to aforementioned elements (abstractions and formalisms), it is characterized as follows.

- a. A wider perspective is offered by abstractions that unify paradigms from the continuous and the discrete world, often in surprising and inspiring ways. The basis is *functional predicate calculus* [15] and *generic functionals* [16]; the latter is the main layer of mathematical concepts complementing logic.
- b. Existing notation is embedded in a general formalism that eliminates ambiguities and inconsistencies, provides useful new forms of expression at no extra cost, and supports formal calculation, also “by hand”. It fully eliminates the style breach in classical mathematics as well as in formal methods.

The unification also entails considerable conceptual simplification. We shall see how the concepts captured by our abstractions are usually known only in various different guises, the similarities hidden by different notations, and properties derived separately for each of these guises. Abstractions allow doing the work once and for all. As observed in [5], *Relief [in coping with monumental growth of usable knowledge] is found in the use of abstraction and generalization [using] simple unifying concepts. This process has sometimes been called “compression”*. This very effective epistemological process also reduces fragmentation.

Overview. The underlying framework is not the main topic here, but a rather terse outline is given in section 2 to provide the wider context.

The main part of the paper deals with abstractions and paradigms, the former being formalized versions of the latter, stripped from their domain-specific

³ Auxiliary tools translating proofs into text are only a shallow patch, making things worse by adding verbosity, not structure. The real issue is a matter of proof style.

connotations. We select some of the most typical unifying abstractions from which the general approach emerged. The topics chosen uncover insights derived from the continuous world which reveal surprising similarities with seemingly disparate discrete concepts, not vaguely, but in a precise mathematical form.

For each topic, we start with a modelling aspect of analog systems, extend it in a direct way to a general abstraction, and then show modelling applications in the discrete world of computing. The first topic (section 3) goes from analog adders and modulators in signal processing and automatic control via a function extension operator to predicate calculus. The second one (section 4) goes from analog filter characteristics via a functional generalization of the Cartesian product to record types, databases and XML semantics. The third one (section 5) goes from distributed systems via lumped ones to program semantics.

Related subjects and ramifications are pointed out along the way. We conclude with notes on the advantages of such far-reaching unifications in the theory and practice of formal methods, tool design, and education in CS and EE.

2 The Basic Formalism as Outlined in the Funmath LRRL

This section explains the formalism used in the sequel. Yet, we shall mostly use our syntax in the conservative mode of synthesizing only common and familiar notations. In this way, most of the notation from section 3 onward will appear entirely familiar, unless (with due warning) the extra power of expression is used.

Hence readers uninterested in formalisms may gloss over section 2, and come back later. Others may also be interested in the wider context. Indeed, the formalism is designed to cover both continuous and discrete mathematics in a formal way with (a) a minimum of syntactic constructs (b) a set of rules for formal calculational reasoning (by hand, as in [21,24,25,26,27]). Therefore this section gives a first idea of how item (a) of this rather ambitious goal is achieved, while item (b) is the main topic of a full course [15]. Since the best compact outline is the “Funmath Language Rationale and Reference Leaflet”, we reproduce here its main portion, taken verbatim (without frills) from an annex to [15].

Rationale. A formal mathematical language is valuable insofar as it supports the design of precise calculation rules that are convenient in everyday practice.

In this sense, mathematical conventions are strong in Algebra and Analysis (e.g., rules for $\int$ in every introductory Analysis text), weaker in Discrete Mathematics (e.g., rules for $\sum$ only in very few texts), and poor in Predicate Logic (e.g., disparate conventions for $\forall$ and $\exists$, rules in most logic texts impractical). This is reflected in the degree to which everyday calculation in these areas can be called “formal”, and inversely proportional to the needs in Computer Science.

Entirely deficient are the conventions for denoting sets. Common expressions such as $\{m \in \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n \in \mathbb{Z}\}$ seem innocuous, but exposing their structure as $\{v \in X \mid p\}$ and $\{e \mid v \in X\}$ (with the metavariables below) reveals the ambiguity: $\{n \in \mathbb{N} \mid n \in \mathbb{Z}\}$ matches both. Calculation rules are nonexistent.

Funmath (Functional Mathematics) is not “yet another computer language” but an approach to structure formalisms by conceiving mathematical objects as functions whenever convenient — which is quite more often than common practice reflects. Four constructs suffice to synthesize most (all?) common conventions without their ambiguities and inconsistencies, and also yield new yet useful new forms of expression, such as point-free expressions. This leaflet covers only syntax and main definitions; calculation rules are the main topic of [15].

Syntax. To facilitate adopting this design in other formalisms, we avoid a formal grammar. Instead, we use metavariables: i for a (tuple of) identifier(s), and for expressions: v, w : (tuple of) variable(s); d, e : arbitrary; p, q, r : boolean; X, Y : set; f, g : function; P, Q : predicate; F, G : family of functions; S, T : family of sets. By “family of X ” we mean “ X -valued function”. Here are the four constructs.

0. An *identifier* can be any (string of) symbol(s) except markers (binding colon and filter mark, abstraction dot), parentheses (), and keywords (**def**, **spec**).
Identifiers are *declared* by *bindings* $i : X \wedge p$, (“ i in X satisfying p ”). The *filter* $\wedge p$ (or **with** p) is optional, e.g., $n : \mathbb{N}$ and $n : \mathbb{Z} \wedge n \geq 0$ are the same.
Definitions, of the form **def** *binding*, declare *constants*, with global scope. Existence and uniqueness are proof obligations, which is not the case for *specifications*, of the form **spec** *binding*. Example: **def** *roto* : $\mathbb{R}_{\geq 0}$ **with** *roto*² = 2. Well-established symbols (e.g., $\mathbb{B}, \Rightarrow, \mathbb{R}, +, \sqrt{}$) are predefined constants.
1. An *abstraction (binding . expression)* denotes a function. The identifiers declared are *variables*, with local scope. Writing f for $v : X \wedge p . e$, the domain axiom is $d \in \mathcal{D}f \equiv d \in X \wedge p[d]_d^v$ and the mapping axiom $d \in \mathcal{D}f \Rightarrow f d = e[d]_d^v$. Here $e[d]_d^v$ is e with d substituted for v . Example: $n : \mathbb{Z} . 2 \cdot n$.
2. A *function application* has the form $f e$ in the default prefix syntax. When binding a function identifier, dashes can specify other conventions, e.g., $\rightarrow$ for infix. Prefix has precedence over infix. Parentheses are used for overriding precedence rules, *never* as an operator. Application may be partial: if $\star$ is an infix operator, then $(a\star)$ and $(\star b)$ satisfy $(a\star) b = a \star b = (\star b) a$. *Variadic application*, of the form $e \star e' \star e'' \star e'''$, is explained below.
3. *Tupling*, of the form e, e', e'' (any length n), denotes a function with domain $0..n-1$ and mapping illustrated by $(e, e', e'') 0 = e$ and $(e, e', e'') 1 = e'$ etc.

Macros can define shorthands in terms of the basic syntax, but few are needed. Shorthands are d^e for $d \uparrow e$ (exponent) and d_e for $d \downarrow e$ (filtering, see below). Sugaring macros are $e \mid v : X \wedge p$ for $v : X \wedge p . e$ and $v : X \mid p$ for $v : X \wedge p . v$, and finally $v := e$ for $v : \iota e$. The *singleton set injector* ι has axiom $d \in \iota e \equiv d = e$.

Functions. A function f is defined by its *domain* $\mathcal{D}f$ and its *mapping* (unique image for every domain element). Skipping a technicality [15], equality is axiomatized by $f = g \Rightarrow \mathcal{D}f = \mathcal{D}g \wedge (e \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f e = g e)$ and its converse, the inference rule $\mathcal{D}f = \mathcal{D}g \wedge (v \in \mathcal{D}f \cap \mathcal{D}g \Rightarrow f v = g v) \vdash f = g$ (new variable v).

Example: the *constant function definer* $\bullet$ with $X \bullet e = v : X . e$ (v not free in e); near-trivial, but very useful. Special instances: the *empty function* $\varepsilon := \emptyset \bullet e$ (any e , by equality) and the *one-point function definer* $\mapsto$ with $d \mapsto e = \iota d \bullet e$.

Predicates are $\mathbb{B}$ -valued functions. Here $\mathbb{B} = \{0, 1\}$, some prefer $\mathbb{B} = \{F, T\}$.

Pragmatics. We show how to exploit the functional mathematics principle and synthesise common notations, issues that are not evident from mere syntax.

(a) *Elastic operators* originally are functionals designed to obviate common ad hoc abstractors like $\sum_{i=m}^n, \forall v: X, \lim_{x \rightarrow a}$, but the concept is more general.

The *quantification* operators ($\forall, \exists$) are defined by $\forall P \equiv P = \mathcal{D}P \bullet 1$ and $\exists P \equiv P \neq \mathcal{D}P \bullet 0$. Observe synthesis of familiar forms in $\forall P \equiv \forall x: \mathcal{D}P. Px$ and $\forall x: \mathbb{R}. x^2 \geq 0$ but also new forms as in $\forall (p, q) = p \wedge q$ and $\exists (p, q) = p \vee q$.

For every infix operator $\star$ an *elastic extension* E is designed such that $x \star y = E(x, y)$. Evident are $\bigcup$ and $\bigcap$ for $\cup$ and $\cap$ ($\mathbf{e.g.}, e \in \bigcap S \equiv \forall x: \mathcal{D}S. e \in Sx$), more interesting are $\sum$ for $+$ (see [15]) and the next extensions for $=$ and $\neq$.

The predicate *con* (*constancy*) with $\text{con } f \equiv \forall x: \mathcal{D}f. \forall y: \mathcal{D}f. fx = fy$ and *inj* (*injectivity*) with $\text{inj } f \equiv \forall x: \mathcal{D}f. \forall y: \mathcal{D}f. fx = fy \Rightarrow x = y$ follow the same design principle. Properties are $\text{con}(d, e) \equiv d = e$ and $\text{inj}(d, e) \equiv d \neq e$.

The (*function*) *range* operator $\mathcal{R}$ has axiom $e \in \mathcal{R}f \equiv \exists x: \mathcal{D}f. fx = e$. Using $\{—\}$ as a synonym for $\mathcal{R}$ synthesizes set notations such as $\{m: \mathbb{N} \mid m < n\}$ and $\{2 \cdot n \mid n: \mathbb{Z}\}$. We never abuse $\in$ for binding, so $\{n: \mathbb{N} \mid n \in \mathbb{Z}\}$ has no ambiguity. Expressions like $\{e, e', e''\}$ also have their usual meaning. Rules are derived via $\exists$. We use $\mathcal{R}$ in defining the *function arrow* $\rightarrow$ by $f \in X \rightarrow Y \equiv \mathcal{D}f = X \wedge \mathcal{R}f \subseteq Y$. For the *partial arrow*, $f \in X \rightarrowtail Y \equiv \mathcal{D}f \subseteq X \wedge \mathcal{R}f \subseteq Y$.

Variadic function application is alternating an infix operator with arguments. We *uniformly* take this as standing for the application of a matching elastic operator to the argument list. Examples: $p \wedge q \wedge r \equiv \forall (p, q, r)$ and $e = e' = e'' \equiv \text{con}(e, e', e'')$. An example of a new opportunity is $e \neq e' \neq e'' \equiv \text{inj}(e, e', e'')$.

Traditional ad hoc abstractors have a “range” attached to them, as in $\sum_{i=m}^n$. Elastic operators subsume this by the domain of the argument. This *domain modulation* principle is supported by the generic *function/set filtering* operator $\downarrow$ defined by $f_P = x: \mathcal{D}f \cap \mathcal{D}P \wedge Px. fx$ and $X_P = \{x: X \cap \mathcal{D}P \mid Px\}$.

(b) *Generic functionals* [16] extend often-used functionals to *arbitrary* functions by lifting restrictions. For instance, function inversion f^- traditionally requires $\text{inj } f$ and composition $f \circ g$ traditionally requires $\mathcal{R}g \subseteq \mathcal{D}f$. We discard all restrictions on the argument functions by defining the domain of the result function such that its image definition is free of out-of-domain applications, e.g., $f \circ g = x: \mathcal{D}g \wedge gx \in \mathcal{D}f. f(gx)$. For the main generic functionals, see [16].

3 From Signal Processing to Predicate Calculus: The Function Extension Paradigm

a. Introduction of the paradigm by example. First, operations defined on instantaneous values are extended formally to signals, in order to express the behaviour of memoryless components. Next, we generalize this by including domain information to obtain a generic functional applicable to all kinds of functions. Thirdly, we illustrate its use in a functional variant of predicate calculus.

(i) The *starting point* is the description of the *behaviour* of certain systems in terms of signals, i.e., functions of type $\mathbb{T} \rightarrow A$ (or $\mathcal{S}_A$) where $\mathbb{T}$ is a time domain and A a set of instantaneous values.

In communications engineering [19] and automatic control [23], the simplest basic blocks are memoryless devices realizing arithmetic operations. Usually the extension of arithmetic operators to signals done implicitly by writing $(x+y)t = xt + yt$. However, we shall see it pays off using an explicit *direct extension* operator $\hat{}$, e.g., for signals x and y with $(x \hat{+} y)t = xt + yt$.

(ii) The *generalization step* consists in making $\hat{}$ generic [16], i.e., applicable to *all* infix operators $\star$ and all functions f and g by suitably defining $f \hat{\star} g$. The criterion for suitability, as motivated in [14], is that the domain for $f \hat{\star} g$ must be defined such that the image definition does not contain any out-of-domain applications. It is easy to see that this requirement is satisfied by defining

$$\begin{aligned} x \in \mathcal{D}(f \hat{\star} g) &\equiv x \in \mathcal{D}f \cap \mathcal{D}g \wedge (fx, gx) \in \mathcal{D}(\star) \\ x \in \mathcal{D}(f \hat{\star} g) &\Rightarrow (f \hat{\star} g)x = (fx) \star (gx). \end{aligned} \quad (1)$$

A noteworthy example is *equality*: $(f \hat{=} g) = x : \mathcal{D}f \cap \mathcal{D}g . fx = gx$, hence $f \hat{=} g$ is a predicate on $\mathcal{D}f \cap \mathcal{D}g$.

(iii) The *particularization step* to applications in predicate and quantifier calculus uses the fact that our predicates are *functions* taking values in $\{0,1\}$. We shall also use the *constant function specifier* $\bullet$: for any set X and any e ,

$$\mathcal{D}(X \bullet e) = X \quad \text{and} \quad x \in X \Rightarrow (X \bullet e)x = e. \quad (2)$$

Our *quantifiers* $\forall$ and $\exists$ are predicates over predicates: for any predicate P ,

$$\forall P \equiv P = \mathcal{D}P \bullet 1 \quad \text{and} \quad \exists P \equiv P \neq \mathcal{D}P \bullet 0. \quad (3)$$

These simple definitions yield a powerful algebra with dozens of calculation rules for everyday practical use [15]. Here we mention only one *theorem* illustrating the role of $\hat{}$, namely $\forall P \wedge \forall Q \Rightarrow \forall(P \hat{\wedge} Q)$. Here is a calculational proof.

$$\begin{aligned} \forall P \wedge \forall Q &\equiv \langle \text{Def. } \forall \rangle \quad P = \mathcal{D}P \bullet 1 \wedge Q = \mathcal{D}Q \bullet 1 \\ &\Rightarrow \langle \text{Leibniz} \rangle \quad \forall(P \hat{\wedge} Q) \equiv \forall(\mathcal{D}P \bullet 1 \hat{\wedge} \mathcal{D}Q \bullet 1) \\ &\equiv \langle \text{Def. } \hat{} \rangle \quad \forall(P \hat{\wedge} Q) \equiv \forall x : \mathcal{D}P \cap \mathcal{D}Q . (\mathcal{D}P \bullet 1)x \wedge (\mathcal{D}Q \bullet 1)x \\ &\equiv \langle \text{Def. } \bullet \rangle \quad \forall(P \hat{\wedge} Q) \equiv \forall x : \mathcal{D}P \cap \mathcal{D}Q . 1 \wedge 1 \\ &\equiv \langle \forall(X \bullet 1) \rangle \quad \forall(P \hat{\wedge} Q) \equiv 1 \end{aligned}$$

This theorem has a conditional converse: $\mathcal{D}P = \mathcal{D}Q \Rightarrow \forall(P \hat{\wedge} Q) \Rightarrow \forall P \wedge \forall Q$.

b. *Some clarifying remarks on predicates.* It is not always customary in logic to view propositions (formulas) as boolean expressions, or predicates as boolean functions. However, this view is common in programming languages, and gives booleans the same status as other types. In fact, we make a further unification with the rest of mathematics, viewing booleans as a restriction of arithmetic to $\{0,1\}$, as in [13]. In a similar approach, Hehner [30] prefers $\{-\infty, +\infty\}$. We chose $\{0,1\}$ because it merges with modulo 2 arithmetic, facilitates counting and obviates characteristic functions in combinatorial and word problems.

Introducing the constants $\{0,1\}$ (or $\{F, T\}$) in traditional logic is somewhat confusing at first, because $p \equiv 1$ is exchangeable with p . This seeming difficulty disappears, however, be pondering the associativity of $\equiv$ in $p \equiv (p \equiv 1)$.

Another issue, raised also by Lamport [35, page 14] is that $x > 1$, taken out of context, can be either a *formula* depending on x , or a (provable) *statement* depending on the hypotheses. In metamathematics, one often uses $\vdash$ to make explicit that provability is meant. However, as Lampson notes, in regular (yet careful) mathematical discourse, symbols like $\vdash$ are not used since the intent (formula versus statement) is clear from the context.

Finally, note that our functional $\forall$ does not “range” over variables but is a predicate (boolean function) over predicates, as in $\forall P$. The familiar variables enter the picture when P is an abstraction of the form $x:X.p$, where p is a formula, so $\forall x:X.p$ has familiar form and meaning. Using *functionals* (like $\forall$) rather than *ad hoc* abstractors (like $\forall x$) is the essence of our elastic operators.

c. Final remarks on direct extension. More recently, the basic concept of direct extension also appears in programming. In the program semantics of Dijkstra and Scholten [21], operators are assumed extended implicitly to structures, e.g., the arithmetic $+$ extends to structures, as in $(x+y).n = x.n + y.n$. This applies even to equality, i.e. if x and y are structures, then $x = y$ does not denote equality of x and y but a function with $(x = y).n \equiv x.n = y.n$. The concept of *polymorphism* in the graphical programming language LabVIEW [6] designates a similar implicit extension. Implicit extension is reasonable in a restricted area of discourse, but it is overly rigid for general practice. An explicit operator offers more flexibility and allows generalization according to our design principles by specifying the result types.

We mention also that *function composition* ($\circ$) is made generic according to the same requirement by defining, for any functions f and g ,

$$\begin{aligned} x \in \mathcal{D}(f \circ g) &\equiv x \in \mathcal{D}g \wedge gx \in \mathcal{D}f \\ x \in \mathcal{D}(f \circ g) &\Rightarrow (f \circ g)x = f(gx). \end{aligned} \tag{4}$$

The (*simplex*) *direct extension* ($\equiv$) for extending single argument functions can now be defined by $\bar{f}g = f \circ g$.

Observe also that, since tuples are functions, $f \circ (x, y) = f x, f y$. This property subsumes the “map” operator for functional programming.

All these operators entail a rich collection of algebraic laws that can be expressed in point-free form, yet preserve the intricate domain refinements (as can be verified computationally). Examples are $f \circ (g \circ h) = (f \circ g) \circ h$ and $\overline{h \circ g} = \bar{h} \circ \bar{g}$ and $(\hat{\star}) = (\overline{\star}) \circ (\&)$. Elaboration is beyond the scope of this paper.

4 From Analog Filters to Record Types: The Function Tolerance Paradigm

a. Introduction of the paradigm by example. Starting with the usual way of specifying the frequency/gain characteristic of a RF filter, we formalize the concept of *tolerance* for functions and generalize it to arbitrary sets. The resulting generic functional, when particularized to discrete domains, subsumes the familiar Cartesian product with a functional interpretation. Combination with

enumeration types expresses record types (in a way quite different from, but “mappable” to, the formulation with projection or selector functions used in Haskell). It is sufficiently general for capturing all types necessary to describe abstract syntax, directory structures, and XML documents. Again we proceed in three steps.

(i) The *starting point* is the specification of analog filter characteristics, for instance gain as a function of frequency. For continuous systems, accuracy of measurements and tolerances on components are an important issue. To extend this notion to functions in a formal way, it suffices to introduce a *tolerance function* T that specifies, for every value x in its domain (e.g., frequency), the set Tx of allowable values (e.g., for the filter gain). More precisely, we say that a function f (e.g., a specific filter characteristic) *meets* the tolerance T iff

$$\mathcal{D}f = \mathcal{D}T \wedge \forall x : \mathcal{D}f \cap \mathcal{D}T . fx \in Tx.$$

This principle, illustrated in Fig. 1, provides the setting for the next two steps.

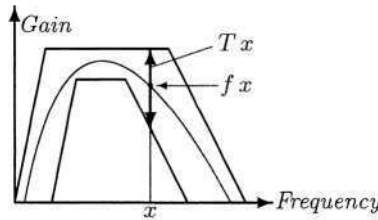


Fig. 1. A bandpass filter characteristic

(ii) The (small) *generalization step* is admitting *any* (not just “dense”) sets for $\mathcal{D}T$. This suggests defining an operator $\times$ such that, if T is a set-valued function, $\times T$ is the set of functions meeting tolerance T :

$$f \in \times T \equiv \mathcal{D}f = \mathcal{D}T \wedge \forall x : \mathcal{D}f \cap \mathcal{D}T . fx \in Tx \quad (5)$$

Observe the analogy with the definition of function equality:

$$f = g \equiv \mathcal{D}f = \mathcal{D}g \wedge \forall x : \mathcal{D}f \cap \mathcal{D}g . fx = gx.$$

With the axiom $x = y \equiv x \in \iota y$ for the *singleton set injector* ι , this yields calculationally $f = g \equiv f \in \times (\iota \circ g)$, hence $\times$ can also specify *exactly*.

(iii) *Particularization step*: Instantiate (5) with $T := A, B$ (two sets). Then

$$f \in \times (A, B) \equiv \mathcal{D}f = \mathbb{B} \wedge f 0 \in A \wedge f 1 \in B$$

by calculation. Hence $\times (A, B) = A \times B$, the usual Cartesian product (considering tuples as functions). This also explains the notation $\times$ and the name *generalized functional Cartesian product* (abbreviated *funcart* product).

As usual, $\times$ defines variadic shorthand for $\times$, as in $A \times B \times C = \times (A, B, C)$. Applied to abstractions, as in $\times a : A . B a$, it covers so-called

dependent types [29], in the literature often denoted by ad hoc abstractions like $\prod_{a \in A} B a$. We also introduce the suggestive shorthand $A \ni a \rightarrow B a$ for $\times a : A. B a$, which is especially convenient in chained dependencies, e.g. $A \ni a \rightarrow B a \ni b \rightarrow C a b$.

b. *Important properties.* In contrast with ad hoc abstractors like $\prod_{a \in A} B a$, the operator $\times$ is a genuine functional and has many useful algebraic properties. Most noteworthy is the inverse. By the axiom of choice, $\times T \neq \emptyset \equiv \forall x : \mathcal{D} T. T \neq \emptyset$. This also characterizes the bijectivity domain of $\times$ and, if $\times T \neq \emptyset$, then $\times^{-}(\times T) = T$. For the usual cartesian product this implies that, if $A \neq \emptyset$ and $B \neq \emptyset$, then $\times^{-}(A \times B) = A, B$, hence $\times^{-}(A \times B) 0 = A$ and $\times^{-}(A \times B) 1 = B$. Finally, an explicit image definition is

$$\times^{-} S = x : \text{Dom } S . \{f x \mid f : S\} \quad (6)$$

for any nonempty S in the range of $\times$, where $\text{Dom } S$ is the common domain of the functions in S (extracted, e.g., by $\text{Dom } S = \bigcap f : S. \mathcal{D} f$).

In fact, the funcart operator is the “workhorse” for typing all structures unified by functional mathematics [12,13]. Obviously, $A \rightarrow B = \times (A \bullet B)$, so it covers all “ordinary” function types as well.

c. *Aggregate data types and structures.* Let $\square n = \{m : \mathbb{N} \mid m < n\}$ for n in $\mathbb{N}' := \mathbb{N} \cup \iota \infty$. For any set A and n in $\mathbb{N}'$, define $A \uparrow n$ (or A^n) by $A \uparrow n = \square n \rightarrow A$, hence $A^n = \times (\square n \bullet A)$, the n -fold product. We also define $A^* = \bigcup n : \mathbb{N}. A^n$.

Apart from sequences, the most ubiquitous aggregate data type are records in the sense of PASCAL [34].

One approach for expressing records functionally is using *selector functions* corresponding to the field labels, where the records themselves appear as arguments. We have explored this alternative some time ago in a different context [8], and it is also currently used in Haskell [33]. However, it does not make records themselves into functions and has a rather heterogeneous flavor.

Therefore our preferred alternative is the $\times$ operator from (5), whereby records are defined as *functions* whose domain is a set of field labels constituting an enumeration type. For instance

$$\text{Person} := \times (\text{name} \mapsto A^* \cup \text{age} \mapsto \mathbb{N}),$$

where *name* and *age* are elements of an enumeration type, defines a function type such that the declaration *employee* : *Person* specifies *employee name* $\in A^*$ and *employee age* $\in \mathbb{N}$. The syntax can be made more attractive by defining, for instance, an elastic type definition operator *Record* with $\text{Record } F = \times (\bigcup F)$, so we can write $\text{Person} := \text{Record } (\text{name} \mapsto A^*, \text{age} \mapsto \mathbb{N})$.

Observe the use of *function merge* $\cup$. A full discussion of this operator [16] is beyond the scope of this paper. However, it suffices to know that, if $f \odot g$, then $\mathcal{D}(f \cup g) = \mathcal{D} f \cup \mathcal{D} g$ and $x \in \mathcal{D} f \Rightarrow (f \cup g) x = f x$ (similarly for g). *Compatibility* for functions is defined by $f \odot g \equiv \forall x \in \mathcal{D} f \cap \mathcal{D} g. f x = g x$.

As mentioned, other structures are also defined as functions. For instance, trees are functions whose domains are *branching structures*, i.e., sets of sequences

describing the path from the root to a leaf in the obvious way. This covers any kind of branch labeling. For instance, for a binary tree, the branching structure is a subset of $\mathbb{B}^*$. Classes of trees are characterized by restrictions on the branching structures. The $\times$ operator can even specify types for leaves individually.

Aggregates defined as functions inherit all elastic operators for which the images are of suitable type. For instance, $\sum s$ sums the fields or leaves of any number-valued record, tree or other structure s .

d. Application to relational databases. Database systems are intended to store information and present a convenient interface to the user for retrieving the desired parts and for constructing and manipulating “virtual tables” containing precisely the information of interest in tabular form.

Code	Name	Instructor	Prerequisites
CS100	Basic Mathematics for CS	R. Barns	MA100 CS100, EE150
MA115	Introduction to Probability	K. Jason	
CS300	Formal Methods in Engineering	R. Barns	
...	...	...	

A *relational database* presents the tables as relations. One can view each row as a tuple, and a collection of tuples of the same type as a relation.

However, in the traditional nonfunctional view of tuples, components can be accessed only by a separate indexing function using natural numbers. This is less convenient than, for instance, the column headings. The usual patch consists in “grafting” onto the relational scheme so-called *attribute names* corresponding to column headings. Disadvantages are that the mathematical model is not purely relational any more, and that operators for handling tables are ad hoc.

Viewing the table rows as *records* in functional sense as before allows embedding in a more general framework with useful algebraic properties and inheriting the generic operators. For instance, the table shown can be declared as $GCI : \mathcal{P} \text{ CID}$, a set of *course information descriptors* whose type is defined by

def $CID := \text{Record} (\text{code} \mapsto \text{Code}, \text{name} \mapsto \mathbb{A}^*, \text{inst} \mapsto \text{Staff}, \text{prreq} \mapsto \text{Code}^*).$

Since in our formalism table rows are functions, queries can be constructed by functionals. As an example, we show how this is done for the most subtle of the usual query constructs in database languages, the (“*natural*”) *join*.

We define the operator $\bowtie$ combining tables S and T by uniting the domains of the elements (i.e., the field names), but keeping only those records for which the same field name in both tables have the same contents, i.e., only *compatible* records are combined. In other words,

$$S \bowtie T = \{s \cup t \mid (s, t) : S \times T \wedge s \text{@} t\}$$

e. Application to XML documents. The following is a simple example of a DTD (Document Type Definition) and a typical instance (here without attributes).

<pre> <!ELEMENT book (title, author*,year?)> <!ELEMENT title (#PCDATA)> <!ELEMENT author (#PCDATA)> <!ELEMENT year (#PCDATA)> </pre>	<pre> <?xml version = "1.0"?> <book> <title>Trees</title> <author>V. Green</author> <year>2003</year> </book> </pre>

The DTD semantics and the instance can be expressed mathematically as follows, with minor tacit simplifications (a complete discussion is given in [46]).

```

booktype = Record (title  $\mapsto \mathbb{A}^*$ , author  $\mapsto (\mathbb{A}^*)^*$ , year  $\mapsto \iota \text{ unspecified} \cup \mathbb{A}^*$ )
bookinst =  $\bigcup$  (title  $\mapsto$  "Trees", author  $\mapsto \tau$  "V. Green", year  $\mapsto$  "2003")

```

The operator τ , for expressing sequences of length 1, is defined by $\tau x = 0 \mapsto x$.

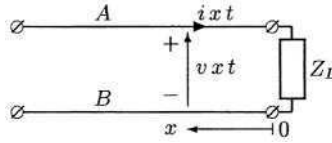


Fig. 2. Introducing the coordinate space paradigm

5 From Transmission Lines to Program Semantics: The Coordinate Space Paradigm

a. Introduction of the paradigm by example. Starting with the well-known telegraphists' equation for transmission lines, we impose a structuring on the parameters involved (voltage, current, time, space), and show how discretization of the space coordinate covers the usual models for lumped circuits and the notion of state in formal semantics. As mentioned earlier, we do this in three steps.

(i) The *starting point* is the typical modelling of dynamical systems in physics and engineering. The example chosen is the simplest model of a transmission line consisting of two wires, with a load at one end, as depicted in Fig. 2. Voltage v and current i , with the conventions shown, are functions of type $\mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ where $\mathbb{S}$ is the *spatial coordinate* space (say, $\mathbb{R}_{\geq 0}$, for the distance from the load), $\mathbb{T}$ is the *temporal coordinate* space ($\mathbb{R}_{\geq 0}$ or $\mathbb{R}$, as desired) and $\mathbb{U}$ is the *instantaneous value* space (voltage, current, typically $\mathbb{R}$). With these conventions, vxt and ixt denote the voltage and the current at location x at time instant t .

Our formulation as higher-order functions (Currying) facilitates defining integral transforms, e.g. for a lossless line, in terms of incident and reflected wave:

$$\begin{aligned}
 vxt &= \vec{v}0(t + x/c) + \overleftarrow{v}0(t - x/c) \\
 Vxf &= \vec{V}0f \cdot e^{j \cdot k_f \cdot x} + \overleftarrow{V}0f \cdot e^{-j \cdot k_f \cdot x}
 \end{aligned}$$

Here $V : \mathbb{S} \rightarrow \mathbb{R} \rightarrow \mathbb{C}$ is the Fourier transform of v with $V x f = \mathcal{F}(v x) f$, and $k_f = 2 \cdot \pi \cdot f/c$. However, it is the time domain formulation $\mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ that provides the setting for the following two steps.

(ii) The (small) *generalization step* consists in admitting *any* (not only dense) coordinates, e.g. refining the above model by introducing a set $W := \{A, B\}$ of names for the wires and new functions for the potential and the current.

$$\begin{aligned} v'_- : W \rightarrow \mathbb{R}_{\geq 0} \rightarrow \mathbb{T} \rightarrow \mathbb{V} \text{ related to } v \text{ by } v'_A x t - v'_B x t &= v x t \\ i'_- : W \rightarrow \mathbb{R}_{\geq 0} \rightarrow \mathbb{T} \rightarrow \mathbb{I} \text{ related to } i \text{ by } i'_A x t &= i x t \wedge i'_B x t = -(i x t) \end{aligned}$$

Discrete coordinates are used in *systems semantics* [10,11] to express the semantics of a language for describing *lumped* adirectional systems [12].

The order of appearance of the space and time coordinates is a matter of convenience. In electronics, one often uses the more “neutral” Cartesian product, writing $v : \mathbb{S} \times \mathbb{T} \rightarrow \mathbb{V}$ and $v(x, t)$. Higher-order functions support 2 variants:

- The *signal space* formulation: a *signal* is a function from $\mathbb{T}$ to $\mathbb{U}$, and quantities of interest are described by functions of type $\mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$.
- The *state space* formulation: a *state* is a function from $\mathbb{S}$ to $\mathbb{U}$, and quantities of interest are described by functions of type $\mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{U}$.

Here $\mathbb{U}$ denotes the universe of values of interest (voltages in our example).

(iii) The *particularization step* to program semantics now simply consists in defining the spatial coordinate space $\mathbb{S}$ to be the set of identifiers introduced in the declarations as variables, $\mathbb{T}$ to be a suitable discrete space for the locus of control (in the syntax tree of the program) and $\mathbb{U}$ the value space specified in the declarations. We define the *state space* $S := \mathbb{S} \rightarrow \mathbb{U}$ or, as a refinement expressing dependence on the type declared for each variable, $S := \mathbb{S} \ni v \rightarrow \mathbb{U}_v$.

For instance, if x, y and z are declared as variables, then $\mathbb{S} = \{x, y, z\}$. We can express the effect of a given command by an equation for the dynamical function $st : \mathbb{T} \rightarrow S$ to express the relationship between the state at the (abstract) time t of execution and the state at time *next* t after execution. E.g., for the assignment

$$'z := x + y'$$

the function $st : \mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{U}$ satisfies the equation⁴

$$st(next\ t) v = (v = z) ? st\ t\ x + st\ t\ y \upharpoonright st\ t\ v. \quad (7)$$

We can eliminate v by two general-purpose axiary operators. *Function over-riding* ($\odot$) is defined as follows. For any functions f and g , the domain of $f \odot g$ is given by $\mathcal{D}(f \odot g) = \mathcal{D}f \cup \mathcal{D}g$, and the image for any x in this domain by $(f \odot g)x = x \in \mathcal{D}f ? g\ x \upharpoonright f\ x$. The operator $\mapsto$ allows writing a function whose domain consists of the single element d , having image e , as $d \mapsto e$. As expected, $((d \mapsto e) \odot g)x = (x = d) ? e \upharpoonright g\ x$ for any x in $\iota e \cup \mathcal{D}g$. Hence (7) becomes

$$st(next\ t) = (z \mapsto st\ t\ x + st\ t\ y) \odot st\ t. \quad (8)$$

⁴ The conditional of the form $c ? b \upharpoonright a$, read: “if c then b else a ”, is self-explanatory.

Since $st(next\ t)$ depends on t only via $st\ t$ (this is the case in general, not just in this example), one can conveniently eliminate time by expressing the effect of every command as a *state transformation*, viz. a function of type $S \rightarrow S$. We combine these functions into one by including the command as a parameter and defining a *meaning function* $\mathcal{C} : C \rightarrow S \rightarrow S$, where C is the set of commands. For instance, the semantics of our assignment is expressed by

$$\mathcal{C} \text{ 'z := x + y' } s = (z \mapsto s\ x + s\ y) \odot s. \quad (9)$$

This is the familiar formulation of denotational semantics [39,48].

Dependence of the locus of control on data can be conveniently expressed by adapting a simple technique from hydrodynamics (relating paths to streamlines) to abstract syntax trees (not elaborated here), whereas environments can be formulated as coordinate transformation.

b. Other applications along the way. Viewing variables as coordinate values raises the question: are they really *variables*? From the unifying viewpoint (and in denotational semantics) they are not: only the *state* varies with time!

This is clarified by the following example, which also illustrates the ramifications for hardware description languages. Consider the digital device (a gate) and the analog device (an operational amplifier) from Fig. 3.

What is the role of the labels $x\ y\ z$ and $x'\ y'\ z'$ on the terminals?



Fig. 3. Devices with labelled terminals

Many choices are possible, for instance

- Names of terminals (just labels in an alphabet).
- Instantaneous values, e.g., boolean ones in $z = x \wedge y$, real ones in $z' = x' - y'$.
- Signals (time functions), e.g., of type $\mathbb{T} \rightarrow \mathbb{B}$ in $z\ t = x\ t \wedge y\ t$ and of type $\mathbb{T} \rightarrow \mathbb{R}$ in $z'\ t = x'\ t + y'\ t$.

Observe that the interpretations are mutually *incompatible*, e.g., the possibility of $x = y$ as *values* conflicts with the obvious fact that $x \neq y$ as *terminals*. Furthermore, using, for instance, x' and y' as function names may require letting $x' = \sin$ for one *gedanken* experiment (or real experiment) and $x' = \cos$ in another. Such “context switching” looks more like assignment in imperative programming than mathematics. Although often harmless, it limits expression.

The way to support all these views without conflict is the coordinate paradigm. In addition to the space coordinate space $\mathbb{S}$, the time coordinate space $\mathbb{T}$ and the instantaneous value space $\mathbb{U}$, we consider an *experiment index space* $\mathbb{Z}$, which supports distinguishing between experiments, but can often be left implicit. As before, we have the signal space formulation $sg : \mathbb{Z} \rightarrow \mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ and the state space formulation $st : \mathbb{Z} \rightarrow \mathbb{T} \rightarrow \mathbb{S} \rightarrow \mathbb{U}$.

In the example of Fig. 3, names of terminals figure as space coordinates. For instance, using the signal space formulation and leaving Z implicit, we make the following typical conventions:

- for the AND-gate: $\mathbb{S} = \{x, y, z\}$ and $\mathbb{T} = \mathbb{N}$ and $\mathbb{U} = \mathbb{B}$
- for the op amp: $\mathbb{S}' = \{x', y', z'\}$ and $\mathbb{T}' = \mathbb{R}$ and $\mathbb{U}' = \mathbb{R}$

The two families of signals $s : \mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ and $s' : \mathbb{S}' \rightarrow \mathbb{T}' \rightarrow \mathbb{U}'$ satisfy respectively

$$s z n = s x n + s y n \quad \text{and} \quad s' z' t = s' x' t + s' y' t. \quad (10)$$

or, equivalently, using *direct extensions*: $s z = s x \hat{\wedge} s y$ and $s' z' = s' x' \hat{+} s' y'$. Direct extension is a topic for later; all that needs to be understood now is that it “extends” an operator $\star$ over instantaneous values to an operator $\hat{\star}$ over signals by $(f \hat{\star} g)t = f t \star g t$. This is such standard practice in communications and control engineering [19,23] that the extension operator ($\hat{}$) is usually left implicit.

c. Defining stochastic processes with the coordinate paradigm. Consider the signal space formulation

$$sg : Z \rightarrow \mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$$

and assume a probability measure on Z is defined

$$Pr : \mathcal{P} Z \rightarrow [0, 1].$$

Then Z (seen as the index set for all possible experiments) will be called a *sample description space* as in [43].

The transpose⁵, of sg , namely sg^T , is of type $\mathbb{S} \rightarrow Z \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ and will be called a *family of stochastic processes*. For every x in $\mathbb{S}$ there is one such process $sg^T x$, of type $Z \rightarrow \mathbb{T} \rightarrow \mathbb{U}$. The distribution function, assuming $\mathbb{U} := \mathbb{R}$, is then defined in the following fashion, where the image definition is the familiar one from the theory of stochastic processes [36,41].

$$\text{def } F_- : \mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U} \rightarrow [0, 1] \text{ with } F_x t \eta = Pr \{ \zeta : Z \mid sg \zeta x t \leq \eta \}$$

For hardwired systems, $\mathbb{S}$ is the set of (real or conceptual) terminals, and the model coincides with the classical one. In program semantics, letting $\mathbb{S}$ be the set of variables and defining a probability measure on Z yields *stochastic semantics*.

c. Functional temporal calculus. We show how the formulation from [9] fits into the coordinate space paradigm. Again letting $sg : Z \rightarrow \mathbb{S} \rightarrow \mathbb{T} \rightarrow \mathbb{U}$ where Z is an index set for all possible experiments, we call the transpose sg^T a *family of temporal variables*. Each temporal variable $sg^T x$ (for given x in $\mathbb{S}$) is a function of type $Z \rightarrow \mathbb{T} \rightarrow \mathbb{U}$. A *temporal operator* is a function of type

$$\mathbb{O} := (Z \rightarrow \mathbb{T} \rightarrow \mathbb{U}) \rightarrow (Z \rightarrow \mathbb{T} \rightarrow \mathbb{W})$$

⁵ Transposition is another generic operator not discussed here; for the special case $f : A \rightarrow B \rightarrow C$, the transpose f^T is of type $B \rightarrow A \rightarrow C$ and satisfies $f^T b a = f a b$.

i.e. from temporal variables to temporal variables. We consider two kinds of temporal operators.

A *temporal mapping* $g:\mathbb{O}$ is defined such that $gv\zeta t = f(v\zeta t)$ for some $f:\mathbb{U} \rightarrow \mathbb{W}$. It can be used to model the input/output behaviour of a *memoryless* system. Typical temporal mappings are *direct extensions* of arithmetic (+, − etc.), propositional ($\wedge$, $\vee$ etc.) and other operators of interest.

A *temporal combinator* is a temporal operator that is not a temporal mapping. Typical temporal combinators are the *next* ($\bigcirc$), *always* ($\Box$) and *sometime* ($\Diamond$) operators defined such that, for every temporal variable $v:Z \rightarrow \mathbb{T} \rightarrow \mathbb{U}$, every $\zeta:Z$ and every $t:\mathbb{T}$ (where $\mathbb{T}$ is assumed ordered) by

$$\begin{aligned}\bigcirc v\zeta t &= v\zeta(\text{next } t) && (\text{e.g., } \text{next } t = t + 1 \text{ for } \mathbb{T} := \mathbb{N}) \\ \Box v\zeta t &= \forall t':\mathbb{T}_{\geq t}. v\zeta t' && (\text{here } \mathbb{U} = \mathbb{W} = \mathbb{B}) \\ \Diamond v\zeta t &= \exists t':\mathbb{T}_{\geq t}. v\zeta t' && (\text{here } \mathbb{U} = \mathbb{W} = \mathbb{B})\end{aligned}$$

Various choices for $\mathbb{T}$ (discrete, continuous, partial orderings, “branching”) and matching definitions are possible, depending on the aspect to be modelled. For instance, for discrete $\mathbb{T}$ with a next operator and $t' \geq t \equiv \exists n:\mathbb{N}. t' = \text{next}^n t$, clearly $\Box v\zeta t = \forall n:\mathbb{N}. v\zeta(\text{next}^n t)$.

Since all these operators are defined as higher-order functions, they can be used in the point-free style with respect to the dummies in Z and $\mathbb{T}$, resulting in expressions of the form (omitting $\sim$)

$$\Box(w \Rightarrow \bigcirc w) \Rightarrow w \Rightarrow \Box w.$$

Hence we can eliminate all explicit reference to time, but also refer to time when systems modelling requires doing so.

The formulas without reference to time are formally identical to those in certain variants of temporal logic [37]. Temporal logic is more abstract in the sense that a given variant may have several models, whereas temporal calculus is a single model in itself. On the other hand, for certain applications pertaining to concrete systems, working within a concrete model may be necessary to keep an explicit relationship with other system aspects. For instance, it is shown in [9] how the next operator $\bigcirc$ can be directly related to the z -transform, an important technique in discrete signal processing and control systems engineering [23,36].

An extension of the functional temporal calculus with a collection of auxiliary functionals [17] is currently being applied to formally specify so-called *patterns* in Bandera [22], a system for modelling concurrent programs written in Java.

6 Concluding Remarks

We have shown how suitable abstractions unify concepts in very disparate fields.

As one referee aptly observed, the mathematics often seems far removed from the area of discourse (e.g., without our explanation, $\times T$ seems unrelated to analog filters) and reasoning amounts to “playing with mathematics”.

This is related to a phenomenon that Wigner calls the “unreasonable effectiveness of mathematics” [47]. For instance, the “mechanical” differential equation $m \cdot D^2 x + k \cdot x = 0$ (for mass m , spring constant k) and the “electrical” one $L \cdot D^2 i + i/C = 0$ (for inductor L , capacitor C) are both captured by the form $a \cdot D^2 f + c \cdot f = 0$. The particulars of the domain of discourse disappear, and one can reason mathematically without distraction by irrelevant concerns.

The “unreasonable effectiveness of mathematics” is directly useful to formal methods. Indeed, rather than designing different methods particular to various application domains (reflecting their idiosyncrasies), unifying models remove irrelevant differences and suggest more generic formal methods whereby the designer can concentrate on exploiting the reasoning power of mathematics.

Admittedly, this goes against the grain of some trends in tool design advocating “being as close as possible to the language of the application domain”. However, it has taken physics a few thousand years to realize the advantages of the opposite way: translating the concepts from the application domain into mathematics (Goethe notwithstanding). Computer science is considerably younger but need not wait a thousand years: once the example has been set, the learning time can be shortened. In fact, much useful mathematics is already there.

Anyway, unification by abstraction provides an important intellectual asset, but when it is also applied to the design of automated tools to support formal methods, it can lead to considerably more commonality and wider scope.

Regarding the preferable *style* of such tools, consider the following calculation examples taken from two typical engineering textbooks, namely [7] and [18].

$$\begin{aligned}
 \frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) l_n(\mathbf{x}) & \qquad F(s) = \int_{-\infty}^{+\infty} e^{-|x|} e^{-i2\pi xs} dx \\
 \leq \frac{1}{n} \sum_{\mathbf{x}} p^n(\mathbf{x}|\theta) [1 - \log q^n(\mathbf{x})] & \qquad = 2 \int_0^{+\infty} e^{-x} \cos 2\pi xs \, dx \\
 = \frac{1}{n} + \frac{1}{n} L(\mathbf{p}^n; \mathbf{q}^n) + H_n(\theta) & \qquad = 2 \operatorname{Re} \int_0^{+\infty} e^{-x} e^{i2\pi xs} dx \\
 = \frac{1}{n} + \frac{1}{n} d(\mathbf{p}^n, \mathcal{G}) + H_n(\theta) & \qquad = 2 \operatorname{Re} \frac{-1}{i2\pi s - 1} \\
 \leq \frac{2}{n} + H_n(\theta) & \qquad = \frac{2}{4\pi^2 s^2 + 1}.
 \end{aligned}$$

The style is *calculational*, and in the second example even purely *equational*, since only equality occurs. Not surprisingly, this is also the most convenient style to formally manipulate expressions in the various unifying system models.

By contrast, tools for logic nearly all use variants of classical formal logic, which amounts to quite different styles such as “natural” reasoning, sequent calculus, tableaux and so on. These styles are acceptable for internal representation but are not suited (nor systematically used) for hand calculation. As pointed out in [26], this seriously hampers their general usefulness. We could add the comment that *the best choice of language or style for any automated tool is one whose formal manipulation rules are convenient for hand calculation as well.*

An aside observation is the following Turing-like test for integration: well-integrated formal methods can formally describe not only target systems but also the concepts and implementations of various tools in a way that is convenient for exposition, *formal* reasoning and proving properties about these tools.

Work by Dijkstra [21], Gries [24,25,26,27] and others shows conclusively that *calculational logic* meets this criterion. Its style is similar to the preceding engineering calculation examples, with $\equiv$ and $\Rightarrow$ as logical counterparts of $=$ and $\leq$. Thereby formal logic becomes a practical tool for everyday use, which explains why it has found wide acceptance in the computing science community during the recent years. Its usefulness would even gain from automated support but, as pointed out in [38], considerable work remains to be done in this direction.

While developing our unifying models, we found calculational logic to merge “seamlessly” with classical algebra and analysis (as used in more traditional physics-based engineering models), thereby closely approximating Leibniz’s ideal. The resulting common ground not only increases the scope, but also considerably lowers the threshold for introduction in industry. This should perhaps be a major consideration in designing tools to support formal engineering methods.

Such a tool could have a core based on substitution and equality (Leibniz’s rule of “equals for equals”) and including function abstraction, surrounded by a layer of prepositional calculus, generic functionals [16] and functional predicate calculus [15], and a second layer implementing mathematical concepts as developed in this paper for unifying system models.

Part of this rationale also underlies B [2]. Differences are our use of generic functions, the functional predicate calculus, and the application to “continuous” mathematics. Similarly, the scope of unification in Hoare and Jifeng’s Unified Theories of Programming [31] is the discrete domain of programming languages.

The concepts presented are also advantageous in education. Factoring out common aspects avoids unnecessary replication, while stimulating the ability to think at a more abstract level. As a fringe benefit, this creates additional room for other topics, which is necessary in view of the rapid technological developments and the limited time available in most curricula, or even the reduction in time as imposed by the Bachelor/Master reform throughout Europe.

References

1. Chritiene Aarts, Roland Backhouse, Paul Hoogendijk, Ed Voermans, Jaap van der Woude, *A relational theory of data types*. Report, Eindhoven University (1992)
2. Jean-Raymond Abrial, *B-Book*. Cambridge University Press (1996)
3. Rajeev Alur, Thomas A. Henzinger, Eduardo D. Sontag, eds., *Hybrid Systems III*, LNCS 1066. Springer-Verlag, Berlin Heidelberg (1996)
4. Henk P. Barendregt, *The Lambda Calculus — Its Syntax and Semantics*, North-Holland (1984)
5. Hyman Bass, “The Carnegie Initiative on the Doctorate: the Case of Mathematics”, *Notices of the AMS*, Vol. 50, No. 7, pp. 767–776 (Aug. 2003)
6. Robert H. Bishop, *Learning with LabVIEW*. Addison Wesley Longman (1999)

7. Richard E. Blahut, *Theory and Practice of Error Control Codes*. Addison-Wesley (1984)
8. Raymond T. Boute, "On the requirements for dynamic software modification", in: C. J. van Spronsen, L. Richter, eds., *MICROSYSTEMS: Architecture, Integration and Use* (Euromicro Symposium 1982), pp. 259-271. North Holland (1982)
9. Raymond T. Boute, "A calculus for reasoning about temporal phenomena", *Proc. NGI-SION Symposium 4*, pp. 405-411 (April 1986)
10. Raymond T. Boute, "System semantics and formal circuit description", *IEEE Transactions on Circuits and Systems*, CAS-33, 12, pp. 1219-1231 (Dec. 1986)
11. Raymond T. Boute, "Systems Semantics: Principles, Applications and Implementation", *ACM TOPLAS* 10, 1, pp. 118-155, (Jan. 1988)
12. Raymond T. Boute, "Fundamentals of Hardware Description Languages and Declarative Languages", in: J. P. Mermet, ed., *Fundamentals and Standards in Hardware Description Languages*, pp. 3-38, Kluwer (1993)
13. Raymond T. Boute, *Funmath illustrated: A Declarative Formalism and Application Examples*. Computing Science Institute, University of Nijmegen (July 1993)
14. Raymond T. Boute, "Supertotal Function Definition in Mathematics and Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 26, No. 7, pp. 662-672 (July 2000)
15. Raymond T. Boute, *Functional Mathematics: a Unifying Declarative and Computational Approach to Systems, Circuits and Programs — Part I: Basic Mathematics*. Course text, Ghent University (2002)
16. Raymond T. Boute, "Concrete Generic Functionals: Principles, Design and Applications", in: Jeremy Gibbons, Johan Jeuring, eds., *Generic Programming*, pp. 89-119, Kluwer (2003)
17. Raymond Boute, Hannes Verlinde, "Functionals for the Semantic Specification of Temporal Formulas for Model Checking", in: Hartmut König, Monika Heiner, Adam Wolisz, eds., *FORTE 2003 Work-in-Progress Papers*, pp. 23-28. BTU Cottbus Computer Science Reports (2003).
18. Ronald N. Bracewell, *The Fourier Transform and Its Applications*, 2nd ed, McGraw-Hill (1978)
19. Ralph S. Carson, *Radio Communications Concepts: Analog*. Wiley (1990)
20. Edmund M. Clarke, O. Gromberg, D. Peled, *Model Checking*. MIT Press (2000)
21. Edsger W. Dijkstra, Carel S. Scholten, *Predicate Calculus and Program Semantics*. Springer (1990)
22. Matthew B. Dwyer, John Hatcliff, *Bandera Temporal Specification Patterns*, <http://www.cis.ksu.edu/santos/bandera/Talks/SFM02/02-SFM-Patterns.ppt>, tutorial presentation at ETAPS'02 (Grenoble) and SFM'02 (Bertinoro), 2002.
23. Gene F. Franklin, J. David Powell, Abbas Emami-Naeini, *Feedback Control of Dynamic Systems*. Addison-Wesley (1986)
24. David Gries, "Improving the curriculum through the teaching of calculation and discrimination", *Communications of the ACM* 34, 3, pp. 45-55 (March 1991)
25. David Gries, Fred B. Schneider, *A Logical Approach to Discrete Math*. Springer (1993)
26. David Gries, "The need for education in useful formal logic", *IEEE Computer* 29, 4, pp. 29-30 (April 1996)
27. David Gries, "Foundations for Calculational Logic", in: Manfred Broy, Birgit Schieder, eds., *Mathematical Methods in Program Development*, pp. 83-126. Springer NATO ASI Series F158 (1997)
28. Robert L. Grossman, Anil Nerode, Anders P. Ravn, Hans Rischel, eds., *Hybrid Systems*, LNCS 736. Springer-Verlag, Berlin Heidelberg (1993)

29. Keith Hanna, Neil Daeche, Gareth Howells, "Implementation of the Veritas design logic", in: Victoria Stavridou, Tom F. Melham, Raymond T. Boute, eds., *Theorem Provers in Circuit Design*, pp. 77–84. North Holland (1992)
30. Eric C. R. Hehner, *From Boolean Algebra to Unified Algebra*. Internal Report, University of Toronto (June 1997, revised 2003)
31. C. A. R. Hoare, He Jifeng, *Unifying Theories of Programming*. Prentice-Hall (1998)
32. Gerard Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley (2003)
33. Paul Hudak, John Peterson, Joseph H. Fasel, *A Gentle Introduction to Haskell 98*. <http://www.haskell.org/tutorial/> (Oct. 1999)
34. Kathleen Jensen, Niklaus Wirth, *PASCAL User Manual and Report*. Springer (1978)
35. Leslie Lamport, *Specifying Systems*, Addison-Wesley (2002).
36. Edward A Lee, David G. Messerschmitt, *Digital Communication* (2nd ed.). Kluwer (1994)
37. Zohar Manna, Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems — Specification*. Springer (1992)
38. Panagiotis Manolios, J. Strother Moore, "On the desirability of mechanizing calculational proofs", *Information Processing Letters*, Vol. 77, No. 2–4, pp. 173–179, (2001)
39. Bertrand Meyer, *Introduction to the Theory of Programming Languages*. Prentice Hall (1991)
40. Sam Owre, John Rushby, Natarajan Shankar, "PVS: prototype verification system", in: P. Kapur, ed., *11th Intl. Conf. on Automated Deduction*, pp. 748–752. Springer Lecture Notes on AI, Vol. 607 (1992)
41. Athanasios Papoulis, *Probability, Random Variables and Atochastic Processes*. McGraw-Hill (1965)
42. Lawrence C. Paulson, *Introduction to Isabelle*, Computer Laboratory <http://www.cl.cam.ac.uk/Research/HVG/Isabelle/dist/docs.html>, University of Cambridge, (Feb. 2001)
43. Emanuel Parzen, *Modern Probability Theory and Its Applications*. Wiley (1960)
44. Frits W. Vaandrager, Jan H. van Schuppen, eds., *Hybrid Systems: Computation and Control*, LNCS 1569. Springer (1999)
45. Moshe Y. Vardi, Pierre Wolper, "An automata-theoretic approach to automatic program verification", *Proc. Symp. on Logic in Computer Science*, pp. 322–331 (June, 1986)
46. Hannes Verlinde, *Systematisch ontwerp van XML-hulpmiddelen in een functionele taal*. M.Sc. Thesis, Ghent University (2003)
47. Eugene Wigner, "The Unreasonable Effectiveness of Mathematics in the Natural Sciences", *Comm. Pure and Appl. Math.* Vol. 13, No. I, pp. 1–14 (Feb. 1960) <http://nedwww.ipac.caltech.edu/level5/March02/Wigner/Wigner.html>
48. Glynn Winskel, *The Formal Semantics of Programming Languages: An Introduction*. MIT Press (1993)

Formally Justifying User-Centred Design Rules: A Case Study on Post-completion Errors

Paul Curzon¹ and Ann Blandford²

¹ Middlesex University, Interaction Design Centre, Bramley Road, London N14 4YZ
p.curzon@mdx.ac.uk

² University College London Interaction Centre, Remax House, 31-32 Alfred Place,
London WC1E 7DP,
a.blandford@ucl.ac.uk

Abstract. Interactive systems combine a human operator with a computer. Either may be a source of error. The verification processes used must ensure both the correctness of the computer component, and also minimize the risk of human error. Human-centred design aims to do this by designing systems in a way that make allowance for human frailty. One approach to such design is to adhere to design rules. Design rules, however, are often ad hoc. We examine how a formal cognitive model, encapsulating results from the cognitive sciences, can be used to justify such design rules in a way that integrates their use with existing formal hardware verification techniques. We consider here the verification of a design rule intended to prevent a commonly occurring class of human error known as the post-completion error.

Keywords: Cognitive architecture, user error, design rules, formal verification.

1 Introduction

Interactive computer systems are systems that combine a human operator with a computer system. Such a system needs to be both correct and usable. With the increasing ubiquity of interactive computer systems, usability becomes increasingly important. Minor usability problems can scale to having major economic and social consequences. Usability has many aspects. We concentrate on one aspect: user error. Humans are naturally prone to error. Such error is not predictable in the way the behaviour of a faulty computer may be. However, much human error is systematic and as such can be modelled and reasoned about.

Design approaches to prevent usability problems often tend to be ad hoc: following lists of design rules, sometimes apparently contradictory, that are based on the experience of HCI experts. Furthermore the considerations of usability experts are often far removed from those of hardware verification approaches, where the emphasis is on correctness of the system against a functional specification. In this paper we consider how the two worlds of formal hardware verification and human-centred usability verification can be integrated. We propose a way in

which usability design rules can be both formalised and derived from formalised principles of cognition within the same framework as hardware verification. We illustrate the approach by considering one well-studied and widely occurring class of systematic human error: the post-completion error. A post-completion error occurs when a user achieves their main goal but omits ‘clean up’ actions; examples include making copies on a photocopier but forgetting to retrieve the original and forgetting to take change from a vending machine.

We first define simple **principles of cognition**. These are principles that generalise the way humans act in terms of the mental attributes of knowledge, tasks and goals. The principles are not intended to be exhaustive, but to cover a variety of classes of cognitive behaviour of interest, based on the motor system, simple knowledge-based cognition, goal-based cognition, *etc.* They do not describe a particular individual, but generalise across people as a class. They are each backed up by evidence from HCI and/or psychology studies. Those presented are not intended to be complete but to demonstrate the approach. We have developed a generic **formal cognitive model** of these principles in higher-order logic. By “generic” we mean that it can be targeted to different tasks and interactive systems. Strictly this makes it a cognitive architecture [16]. In the remainder of the paper we will refer to the generic model as a *cognitive architecture* and use the term *cognitive model* for a version of it instantiated for a given task and system. The underlying principles of cognition are formalised once in the architecture, rather than having to be re-formalised for each new task or system of interest. Whilst higher-order logic is not essential for this, its use makes the formal specifications simpler than the use of a first-order logic would.

The principles, and more formally the cognitive architecture, specify **cognitively plausible behaviour** (see [7]). That is, they specify possible traces of user actions that can be justified in terms of the specific principles. Of course users might also act outside this behaviour, about which situations the model says nothing. Its predictive power is bounded by the situations where people act according to the principles specified. All theorems in this paper are thus bounded by that assumption. That does not preclude useful results from being obtained, provided their scope is remembered. The architecture allows us to investigate what happens if a person does act in such plausible ways. The behaviour defined is neither “correct” nor “incorrect”. It could be either depending on the environment and task in question. It is, rather, “likely” behaviour. We do not model erroneous behaviour explicitly. It emerges from the description of cognitively plausible behaviour. The focus of the description is on the internal goals and knowledge of a user. This contrasts with a description of a user’s actions as, say, a finite state machine that makes no mention of such cognitive attributes.

After describing the architecture, we next formalise a particular class of systematic user error, that is made in a wide range of situations, in terms of the cognitive architecture. We also formalise a simple and well known usability design rule that, if followed, eliminates this class of error. We prove a theorem that states that if the design rule is followed, then the erroneous behaviour cannot

occur *due to the specified cause* as a result of a person behaving according to the principles of cognition formalised.

The design rule is initially formalised in user-centred terms. To enable the integration with machine-centred verification, we next reformulate it in a machine-centred way, ultimately proving that a machine-centred version of the design rule implies the absence of the class of error considered. Even though the cognitive architecture is capable of making the error, the design rule ensures that the user environments (as provided by the computer part of the system) in which it would emerge do not occur. Other errors are, of course, still possible. The main contribution of this paper is to demonstrate a way that formal reasoning about design rules can be achieved based on a cognitive architecture but within the same framework as verification of other aspects.

We have used the HOL interactive proof system [15] so theorems are machine-checked. Given the relative simplicity of the theorems this is not essential in that hand proofs alone would have been possible. Machine-checked proof does give an extra level of assurance over that of the informal proofs upon which they are based. Furthermore our work sets out a framework in which these theorems can be combined with complex machine-checked hardware verification. Machine-checking of the design rule proofs maintains a consistent treatment. Finally, this work aims to demonstrate a general approach. For more complex design rules, the proofs may be harder so machine-checking may be more directly useful.

2 Related Work

There are several approaches to formal reasoning about the usability of interactive systems. One approach is to focus on a formal specification of the user interface [9]. Most commonly it is used with model-checking-based verification; investigations include whether a given event can occur or whether properties hold of all states. In contrast, Bumbulis *et al* [5] verified properties of interfaces based on a guarded command language embedded in the HOL system. Back *et al* [1] illustrate how properties can be proved and data refinement performed of a specification of an interactive system. However, techniques that focus on the interface do not directly support reasoning about design problems that lead to users making systematic errors; also, the usability properties checked are necessarily device-specific and have to be reformulated for each system verified.

An alternative is formal user modelling of the underlying system. It involves writing both a formal specification of the computer system and one of the user, to support reasoning about their conjoint behaviour. Both system and user are considered as central components of the system and modelled as part of the analysis. Doing so provides a conceptually clean method of bringing usability concerns into the domain of traditional verification in a consistent way. Duke *et al* [13] express constraints on the channels and resources within an interactive system; this approach is particularly well suited to reasoning about interaction that, for example, combines the use of speech and gesture. Moher and Dirda [21] use Petri net modelling to reason about users' mental models and their changing

expectations over the course of an interaction; this approach supports reasoning about learning to use a new computer system but focuses on changes in user belief states rather than proof of desirable properties. Paterno' and Mezzanotte [22] use LOTOS and ACTL to specify intended user behaviours and hence reason about interactive behaviour.

Our work complements these uses of formal user modelling. None of the above focus on reasoning about user errors. Models typically describe how users are intended to behave: they do not address human fallibility. If verification is to detect user errors, a formal specification of the user, unlike one of a computer system, is not a specification of the way a user should be; rather, it is a description of the way they are [7]. Butterworth *et al* [6] do take this into account, using TLA to reason about reachability conditions within an interaction. Rushby [25] formalised plausible mental models of systems, looking for discrepancies between these and actual system behaviour. However, like interface-oriented approaches, each model is individually hand-crafted for each new device in this work.

An approach to interactive system verification that focuses directly on errors is exemplified by Fields [14]. He models erroneous actions explicitly, analysing the consequences of each possible action. He thus models the effect of errors rather than their underlying causes. A problem of this approach is the lack of discrimination about which errors are the most important to consider. It does not discriminate random errors from systematic errors which are likely to re-occur and so be most problematic. It also implicitly assumes there is a "correct" plan, from which deviations are errors.

The University of Queensland's safeHCI project [20] has similar aims and approach to our overall project, combining the areas of cognitive psychology, human-computer interaction and system safety engineering. The details differ, however. SafeHCI has had a focus on hazard analysis and system-specific modelling, whereas our work has an emphasis on generic cognitive models.

Approaches that are based on a cognitive architecture (e.g. [19] [17] [23]) model underlying cognitive causes of errors. However, the modelling exemplified by these approaches is too detailed to be amenable to formal proof. Our previous work [11] followed this approach but at a coarser level of detail, making formal proof tractable. In this approach general mechanisms of cognition are modelled and so need be specified only once, independent of any given interactive system. Furthermore, by explicitly doing the verification at the level of underlying cause, on failed verification, a much greater understanding of the problem is obtained. Rather than just knowing the manifestation of the error – the actions that lead to the problem – the failed proof provides understanding of the underlying causes.

Blandford *et al* [4] have used a formal model of user behaviour to derive high level guidance. There the emphasis is on a semi-formal basis underpinning the craft skill in spotting when a design has usability problems. We are concerned here with guidance for a designer rather than for a usability analyst. We focus on the verification of general purpose design rules rather than the interactive systems themselves.

```

USER flag actions commitments commgoals init_commgoals stimulus_actions
    possessions finished finishedpos goalachieved invariant
    (ustate:'u) (mstate:'m) =
    (USER_CHOICE flag actions commitments commgoals stimulus_actions
        finished finishedpos goalachieved invariant ustate mstate) ∧
    (USER_UNIVERSAL actions commgoals possessions finished flag ustate mstate)

USER_CHOICE flag actions commitments commgoals stimulus_actions
    finished finishedpos goalachieved invariant ustate mstate =
(∀t.
    ¬(flag t) ∨
    (IF (finished ustate (t-1))
        THEN (NEXT flag actions finishedpos t)
        ELSE IF ((CommitmentMade (CommitmentGuards commitments) t)
            THEN (COMMITTS flag actions commitments t)
            ELSE IF TASK_DONE (goalachieved ustate) (invariant ustate) t
            THEN (NEXT flag actions finishedpos t)
            ELSE USER_RULES flag actions commgoals stimulus_actions finishedpos
                goalachieved mstate ustate t))))

USER_RULES flag actions commgoals stimulus_actions finishedpos goalachieved
    (mstate:'m) (ustate:'u) t =
    COMPLETION flag actions finishedpos goalachieved ustate t ∨
    REACTS flag actions stimulus_actions t ∨
    COMMGOALER flag actions commgoals goalachieved ustate mstate t ∨
    ABORTION flag actions finishedpos goalachieved commgoals stimulus_actions
        ustate mstate t

```

Fig. 1. The USER relation

Providing precision to ensure different people have the same understanding of a concept has been suggested as the major benefit of formal models in interaction design [3]. One approach would therefore be to just formalise the design rules (see [3], [24]). In our approach, we not only formalise design rules, we also prove theorems justifying them based on underlying principles about cognition embodied in a formal cognitive architecture. In this way the design rules are formally demonstrated to be correct, up to the assumptions of the principles of cognition. This gives extra assurance to those applying the design rules. This approach builds on our previous work where informal argument only was used to justify the effectiveness of design rules [12]. We show here how this can be formalised in the same framework as other forms of verification.

3 Formalising Cognitively Plausible Behaviour

We first describe our cognitive architecture. It is specified by a higher-order logic relation USER, the top levels of which are given in Figure 1. It takes as

arguments information such as the user's goal, goalachieved, a tuple of actions that the user may take, actions, *etc.* The final two arguments, *ustate* and *mstate*, each of polymorphic type as specified by the type variables 'u and 'm, represent the user state and the machine state over time. The specific type is only given when the architecture is instantiated for a given interaction. These states record over time the series of mental and physical actions made by the user, together with a record of the user's possessions. They are instantiated to a tuple of history functions: functions of type $\text{time} \rightarrow \text{bool}$, from time instances to a boolean indicating whether that signal is true at that time (i.e. the action is taken, the goal is achieved, etc). The other arguments to *USER* specify accessor functions to one of these states. For example, *finished* is of type $'u \rightarrow \text{time} \rightarrow \text{bool}$. Given the user state it returns a history function that for each time instance indicates whether the user model has terminated the interaction. The other arguments of the model will be examined in more detail as needed in the explanation of the model below.

The *USER* relation is split into two parts. The first, *USER_CHOICE*, models the user making a choice of actions. It formalises the action of the user at a given time as a series of rules, one of which is followed at each time instance. *USER_UNIVERSAL* specifies properties that are true at all time instances, whatever the user does. For example, it specifies properties of possessions such that if an item is not given up then the user still has it. We focus here on the choice part of the model as it is most relevant to the concerns of this paper. *USER_CHOICE* is therefore described in detail below. In outline, it states that the next user action taken is determined as follows:

if the interaction is finished
then it should remain finished
else if a physical action was previously decided on
then the physical action should be taken
else if the whole task is completed
then the interaction should finish
else an appropriate action should be chosen non-deterministically

The cognitive architecture is ultimately, in the final else case above, based on a series of non-deterministic temporally guarded action rules, formalised in relation *USER_RULES*. Each describes an action that a user *could* plausibly make. The rules are grouped (e.g. in definition *REACTS* in Figure 1) corresponding to a user performing actions for specific cognitively related reasons. Each such group then has a single generic description. Each rule combines a pre-condition such as a particular message being displayed, with an action, such as a decision made to press a given button at some later time.

rule 1 fires asserting its action is taken $\vee$
 rule 2 fires asserting its action is taken $\vee$
 ...
 rule n fires asserting its action is taken

Apart from those included in the if-then-else staircase of `USER_CHOICE`, no further priority ordering between rules is modelled. We are interested in whether an action is cognitively plausible at all (so could be systematically taken), not whether one is more likely than another. We are concerned with design rules that prevent any systematic erroneous action being taken even if in a situation some other action is more likely anyway. The architecture is a relation. It does not assert that a rule will be followed, just that it may be followed. It asserts that the behaviour of any rule whose guards are true at a point in time is cognitively plausible at that time. It cannot be deduced that any specific rule will be the one that the person will follow if several are cognitively plausible.

The architecture is based on a temporal primitive, `NEXT` that specifies the next user action taken *after* a given time. `NEXT` flag actions `action t` states that the `NEXT` action performed *after time* `t` from a list of all possible user actions, `actions`, is `action`. It asserts that the given action's history function is true at some first point in the future, and that the history function of all other actions is false up to that point. The action argument is of type integer and specifies the position of the action history function in the list `actions`. The flag argument to `NEXT` and `USER` is a specification artifact used to ensure that the time periods that each firing rule specifies do not overlap. It is true at times when a new decision must be made by the model. The first line of `USER_CHOICE` in Figure 1 thus ensures, based on the truth of the flag, that we do not re-specify contradictory behaviour in future time instances to that already specified. Consider the first if-then-else statement of `USER_CHOICE` in Figure 1 as an example of the use of `NEXT`. The action argument of `NEXT` is instantiated to `finishedpos`. It states that if the interaction was finished then the next action remains finished: once the interaction has terminated the user takes no other action.

We model both **physical and mental actions**. A person decides (making a mental action) to take a physical action before it is actually taken. Once a signal has been sent from the brain to the motor system to take the physical action, the signal cannot be revoked even if the person becomes aware that it is wrong before the action is taken. Each physical action modelled is thus associated with an internal mental action that commits to taking it. The argument `commitments` to the relation `USER` is a list of pairs that links the mental and physical actions. `CommitmentGuards` extracts a list of all the mental actions (the first elements of the pairs). The recursively defined, `CommitmentMade` checks, for a given time instance, `t`, if any mental action was taken in the previous time instance (`cmt (t-1)`):

$$(\text{CommitmentMade } [] \ t = \text{FALSE}) \wedge \\ (\text{CommitmentMade } (\text{cmt} :: \text{rest}) \ t = (\text{cmt}(t-1)) \vee (\text{CommitmentMade } \text{rest } t))$$

If a mental action, `mact`, made a commitment to a physical action `pact` on the previous cycle (time, `t-1`) then that will be the next action taken. Definition `COMMITTS` asserts this disjunctively for the whole list of commitments:

COMMIT flag actions $\text{mact pact } t = (\text{mact } (t-1)) \wedge \text{NEXT flag actions pact } t$
 $(\text{COMMITTS flag actions } \square t = \text{FALSE}) \wedge$
 $(\text{COMMITTS flag actions } (l :: \text{commits\_actions}) t =$
 $((\text{COMMITTS flag actions } \text{commits\_actions } t) \vee$
 $(\text{COMMIT flag actions } (\text{CommitmentGuard } l) (\text{CommitmentAction } l) t)))$

Based on these definitions the second if statement of `USER_CHOICE` in Figure 1 states that if a mental action is taken on a cycle then the next action will be the externally visible action it committed to. The physical action already committed to by a mental action is thus given high priority as modelled by being in the if-then-else staircase.

Task-based termination behaviour: In the third if statement of definition `USER_CHOICE` it specifies that a user will terminate an interaction when their whole task is achieved. The user has a goal and the task is not completed until that goal is achieved. We must therefore supply a relation argument `goalachieved` to the cognitive architecture that indicates over time whether the goal is achieved or not. With a vending machine, for example, this may correspond to the person's possessions including chocolate. Similar to `finished`, `goalachieved` extracts from the state a history function that, given a time, returns a boolean value indicating whether the goal is achieved at that time. Note that `goalachieved` is a higher-order function and can as such represent an arbitrarily complex condition. It might, for example, be that the user has a particular object as above, that the count of some series of objects is greater than some number or a combination of such atomic conditions.

In achieving a goal, subsidiary tasks are often generated. For the user to complete the task associated with their goal they must also complete all subsidiary tasks. The underlying reason for these tasks being performed is that in interacting with the system some part of the state must be temporarily perturbed in order to achieve the desired task. Before the interaction is completed such perturbations must be undone. Examples of such tasks with respect to a vending machine include taking change. One way to specify these tasks would be to explicitly describe each such task. Instead we use the more general concept of an interaction invariant [11]: a higher-order argument to the cognitive architecture. The interaction invariant is an invariant at the level of abstraction of whole interactions in a similar sense to a loop invariant in program verification. For example, the invariant for a simple vending machine might be true when the total value of the user's possessions (coins and chocolate) have been restored to their original value, the user having exchanged coins for chocolate of the same value. Task completion involves not only completing the user's goal, but also restoring the invariant.

`TASK_DONE` `goalachieved` invariant $t = (\text{goalachieved } t \wedge \text{invariant } t)$

We assume that on completing the task in this sense, the interaction will be considered terminated by the user unless there are physical actions already committed to. It is therefore modelled in the if-then-else staircase of `USER_CHOICE` to give it priority over other rules apart from committed actions.

We next examine the non-deterministic rules in the final else case of definition `USER_CHOICE` that form the core of the model and are defined in `USER_RULES`.

COMPLETION: Cognitive psychology studies have shown that users intermittently, but persistently, terminate interactions as soon as their goal has been achieved [8]. This behaviour is formalised as a guarded rule. If the goal is achieved at a time then the next action of the cognitive architecture can be to terminate the interaction:

`COMPLETION` flag actions finished goalachieved (ustate:'u) t =
(goalachieved ustate t) $\wedge$ `NEXT` flag actions finished t

REACTS: A user may react to a stimulus from a device, doing the action suggested by it. For example, if a flashing light comes on next to the coin slot of a vending machine, a user might, if the light is noticed, react by inserting coins. In a given interaction there may be many different stimuli to react to. Rather than specify this behaviour for each, we define it generically. Relation `REACT` gives the rule defining what it means to react to a given stimulus. If at time t , the stimulus `stim` is active, the next action taken by the user out of possible actions, actions, at an unspecified later time, may be the associated action.

`REACT` flag actions stim action t = stim t $\wedge$ `NEXT` flag actions action t

As there may be many reactive signals, the user model is supplied with a list of stimulus-action pairs: [(s1, a1); ... (sn, an)]. `REACTS`, given a list of such pairs, recursively extracts the components and asserts the above rule about them. The clauses are combined using disjunction, so are non-deterministic choices, and this definition is combined with other non-deterministic rules. `Grd` and `Act` extract a pair's components.

(`REACTS` flag actions [] t = FALSE) $\wedge$
(`REACTS` flag actions (s :: st) t =
((`REACTS` flag actions st t) $\vee$ (`REACT` flag actions (`Grd` s) (`Act` s) t)))

COMMGOALER: A user often enters an interaction with knowledge of the task, if not the device used to achieve it. They may, as a result, start with sub-goals that they know must be discharged to achieve their main goal. This kind of preconceived sub-goal is known as a *communication goal* [2]. For example, when the user has the goal of purchasing a ticket, they are likely to know that in some way the destination and ticket type must be specified as well as payment made. Communication goals are distinct from device dependent sub-goals that result from the person reacting to stimulus from the device or "tidying" sub-goals that restore a perturbation made to the device from the initial state. The precise nature of the action associated with a communication goal may not be known in advance. A communication goal specification is not a fully specified plan, in that no order of the corresponding actions may be specified. The way that these must be done and their order may not be known in advance. If the person sees an apparent opportunity to discharge a communication goal they

may do so. Once they have done so they will not expect to need to do so again. No fixed order is assumed over how communication goals will be discharged if their discharge is apparently possible. Communication goals are a reason why people do not just follow instructions.

We model communication goals as guard-action pairs as for reactive signals. The guard describes the situation under which the discharge of the communication goal appears possible, such as when a virtual button actually is on the screen. As for reactive behaviour, the architecture is supplied with a list of (guard, action) pairs one for each communication goal. Unlike the reactive signal list that does not change through an interaction, communication goals are discharged. This corresponds to them disappearing from the user's mental list of intentions. We model this by removing them from the communication goal list when done. We do not go into detail of the formalisation of communication goals here as it is not directly relevant. The interested reader should see [11].

ABORTION: A user may terminate an interaction when there is no apparent action they can take that would help complete the task. For example, if on a touch screen ticket machine, the user wishes to buy a weekly season ticket, but the options presented include nothing about season tickets, then the person might give up, assuming their goal is not achievable. The model includes a final default non-deterministic rule, **ABORTION**, that models this case by just forming the negation of the guards of all other rules.

The features of the cognitive architecture discussed above concern aspects of cognition. An extension of the architecture for this paper over that of our previous work [11] as given in Figure 1 involves the addition of **probes**. Probes are extra signals that do not alter the cognitive behaviour of the architecture, but instead make internal aspects of its action visible. This allows specifications to be written in terms of hidden internal cognitive behaviour, rather than just externally visible behaviour. This is important for this work as our aim is to formally reason about whether design rules address underlying cognitive causes of errors not just their physical manifestation. The form of probe we consider here records for each time instance whether a particular rule fires at that instance. We require a single probe that fires when the goal-based termination rule described above fires. We formalise this using a function, `Goalcompletion` that extracts the goal completion probe from the collection of probes passed as an additional argument to the cognitive architecture. To make the probe record goal completion rule events, we add a clause specifying the probe is true to the rule concerning goal completion, **COMPLETION** given above:

$(\text{Goalcompletion probes } t) \wedge \text{goalachieved } t \wedge \text{NEXT flag actions finished } t$

Each other rule in the architecture has a clause added asserting the probe is false at the time it fires. For example the **REACT** rule becomes:

$(\text{Goalcompletion probes } t = \text{FALSE}) \wedge \text{stim } t \wedge \text{NEXT flag actions action } t$

A similar clause is also added to the part of the architecture that describes the behaviour when no rule is firing.

4 Verifying a User Error Design Rule

Erroneous actions are the immediate, obvious cause of failure attributed to human error, as it was a particular action (or inaction) that caused the problem: users pressing a button at the wrong time, for example. However, to understand the problem, and so minimize re-occurrence, approaches that consider the immediate causes alone are insufficient. It is important to consider *why* the person took that action. The ultimate causes can have many sources. Here we consider situations where the ultimate causes of an error are that limitations of human cognition have not been addressed in the design. An example might be that the person pressed the button at that moment because their knowledge of the task suggested it would be sensible. Hollnagel [18] distinguishes between human error phenotypes (classes of erroneous actions) and genotypes (the underlying psychological cause). He identifies a range of simple phenotypes such as repetition of an action, omission of actions, etc. In this paper, to demonstrate the feasibility of formally reasoning about design rules based on cognitively plausible behaviour, we consider one particular error genotype: the class of errors known as *post-completion errors* introduced in Section 1. A similar effect (i.e. phenotype) to a post completion error can occur for other reasons. However that would be considered a different class of error (genotype). Other design rules might be required to prevent it.

4.1 Formalising Post-completion Error Occurrence

In our cognitive architecture post completion error behaviour is modelled by the goal termination rule firing. Probe signal `Goalcompletion` records whether that particular rule has fired at any given time. Note that the rule can fire when the goal is achieved but does not have to. Note also that it firing is necessary but not sufficient for the cognitive architecture to make a post-completion error. In some situations it is perfectly correct for the rule to fire. In particular if the interaction invariant has been re-established at the point when it fires then an error has not occurred. Thus whilst the error occurring is a direct consequence of the existence of this rule in the model, the rule is not directly modelling erroneous actions, just cognitively plausible behaviour that leads to an erroneous action in some situations.

Definition `PCE_OCCURS` specifies that a post-completion error occurs if there is a time, t , before the end time of the interaction t_e , such that the probe `Goalcompletion` is true at that time but invariant has not been re-established.

`PCE_OCCURS` probes invariant $t_e =$

$$(\exists t. t \leq t_e \wedge \text{Goalcompletion probes } t \wedge \neg(\text{invariant } t))$$

This takes two higher order arguments, representing the collection of probes indicating which rules fire and the relation indicating when the interaction invariant is established. A final argument indicates the end time of interest. It bounds the interaction under consideration corresponding to the point when the

user has left and the machine has reset. The start time of the interaction is assumed to be time zero.

4.2 Formalising a Design Rule

We next formalise a well-known user-centred design rule intended to prevent a user having the opportunity to make a post-completion error. It is based on the observation that the error occurs because it is possible for the goal to be achieved before the task as a whole has been completed. If the design is altered so that all user actions have been completed before the goal then a post-completion error will not be possible. In particular any tidying up actions associated with restoring the interaction invariant must be either done by the user before the goal can possibly be achieved, or done automatically by the system. This is the design approach taken for British cash machines where, unlike in the original versions, cards are always returned before cash is dispensed. This prevents the post-completion error where the person takes the cash (achieving their goal) but departs without the card (a tidying task).

The formal version of the design rule states that for all times less than the end time, te , it is not the case that both the goal is achieved at that time and the task is not done. Here, `goalachieved` and `invariant` are the same as in the cognitive architecture.

PCE_DR goalachieved invariant $te =$
 $(\forall t. t \leq te \supset \neg(\text{goalachieved } t \wedge \neg(\text{TASK\_DONE goalachieved invariant } t)))$

Thus when following this design approach, the designer must ensure that at all times prior to the end of the interaction it is not the case that the goal is achieved when the task as a whole is incomplete. The design rule was formulated in this way to match a natural way to think about it informally according to the above observation.

4.3 Justifying the Design Rule

We now prove a theorem that justifies the correctness of this design rule (up to assumptions in the cognitive architecture). If the design rule works, at least for users obeying the principles of cognition, then the cognitive architecture's behaviour when interacting with a machine satisfying the design rule should never lead to a post-completion error occurring. We have proved using HOL the following theorem stating this:

$\vdash \text{USER} \dots \text{goalachieved invariant probes} \text{ustate mstate} \wedge$
 $\text{PCE\_DR} (\text{goalachieved} \text{ustate}) (\text{invariant} \text{ustate}) te \supset$
 $\neg(\text{PCE\_OCCURS probes} (\text{invariant} \text{ustate}) te)$

We have simplified, for the purposes of presentation the list of arguments to the relation `USER` which is the specification of the cognitive architecture, omitting those arguments that are not directly relevant to the discussion. One way

to interpret this theorem is as a traditional correctness specification against a requirement. The requirement (conclusion of the theorem) is that a post-completion error does not occur. The conjunction of the user and design rule is a system implementation. The system is implemented by placing an operator (as specified by the cognitive architecture USER) with the machine (as minimally specified by the design rule). The definitions and theorem proved are generic. They do not specify any particular interaction or even task. A general, task independent design rule has thus been verified.

The proof of the above theorem is simple. It involves case splits on the goal being achieved and the invariant being established. The only case that does not follow immediately is when the goal is not achieved and the invariant does not hold. However, this is inconsistent with the goal completion rule having fired so still follows fairly easily.

4.4 Machine-Centred Rules

The above design rule is in terms of user concerns – an invariant of the form suitable for the cognitive model and a user-centred goal. Machine designers are not directly concerned with the user and this design rule is not in a form that is directly of use. The designer cannot manipulate the user directly, only machine events. Thus whilst the above rule and theorem are in a form of convenience to a usability specialist, they are less convenient to a machine designer. We need a more machine-centred design rule as below.

MACHINE_PCE_DR $\text{goalevent } t \supset (\forall t1. t \leq t1 \wedge t1 \leq te \supset \text{minvariant } t1))$

This design rule is similar to the user-centred version, but differs in several key ways. Firstly, the arguments no longer represent user based relations. The `goalevent` signal represents a machine event. Furthermore this is potentially an instantaneous event, rather than a predicate that holds from that point on. Similarly, the machine invariant concerns machine events rather than user events. Thus, for example with a vending machine, the goal as specified in a user-centred way is that the user has chocolate. Once this first becomes true it will continue to hold until the end of the interaction, since for the purposes of analysis we assume that the user does not give up the chocolate again until after the interaction is over. The machine event however, is that the machine fires a signal that releases chocolate. This is a relation on the machine state rather than on the user state: `GiveChoc mstate`. It is also an event that occurs at a single time instance (up to the granularity of the time abstraction modelled). The machine invariant is also similar to the user one but specifying that the value of the *machine's* possessions are the same as at the start of the interaction – it having exchanged chocolate for an equal amount of money. It is also a relation on the machine's state rather than on the user's state.

The ramification of the goal now being an instantaneous event is that we need to assert more than that the invariant holds whenever the goal achieved

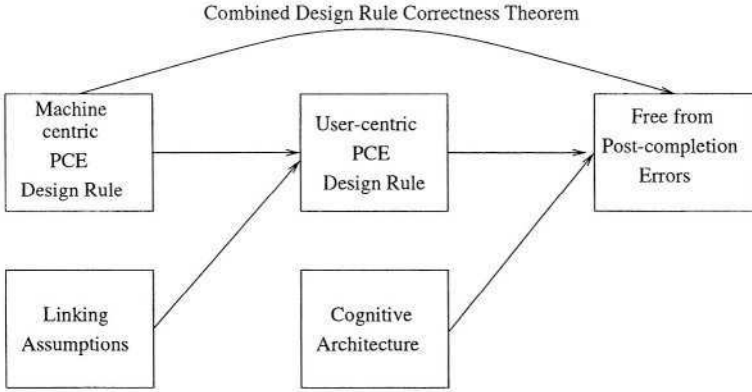


Fig. 2. Verifying the Design Rule in Stages

event holds. The invariant must hold from that point up to the end of the interaction. That is the reason a new universally quantified variable t_1 appears in the definition, constrained between the time the goal event occurs and the end of the interaction.

We prove that this new design rule implies the original, provided assumptions are met about the relationship between the two forms of goal statements and invariants. It is these assumptions that form the basis of the integration between the user and machine-centred worlds.

$$\begin{aligned}
 &\vdash (\forall t. \text{minvariant } t \supset \text{invariant } t) \wedge \\
 &\quad (\forall t. (\text{goalachieved } t) \supset \exists t_2. t_2 \leq t \wedge (\text{goalevent } t_2)) \supset \\
 &\quad \text{MACHINE\_PCE\_DR } \text{goalevent } \text{minvariant } t \supset \\
 &\quad \text{PCE\_DR } \text{goalachieved } \text{invariant } t
 \end{aligned}$$

This asserts that the machine based design rule MACHINE_PCE_DR does indeed imply the user-centred one PCE_DR, under two assumptions. The first assumption is that at all times the machine invariant being true implies that the user invariant is true at that time. The second assumption asserts a connection between the two forms of goal statement. If the user has achieved their goal at some time t then there must have existed an earlier time t_2 at which the machine goal event occurred. The user cannot achieve the goal without the machine enabling it.

4.5 Combining the Theorems

At this point we have proved two theorems. Firstly we have proved that a machine-centred statement of a design rule implies a user-centred one, and secondly that the user-centred design rule implies that post-completion errors are not made by the cognitive architecture. These two theorems can be combined giving us a theorem that justifies the correctness of the machine-centred design rule with respect to the occurrence of post-completion errors as illustrated in Figure 2. The theorem proved in HOL is:

$$\begin{aligned}
&\vdash (\forall t. \text{minvariant } t \supset \text{invariant ustate } t) \wedge \\
&\quad (\forall t. (\text{goalachieved } t) \supset \exists t2. t2 \leq t \wedge (\text{goalevent } t2)) \supset \\
&\quad \text{MACHINE\_PCE\_DR goalevent minvariant te} \wedge \\
&\quad \text{USER} \dots \text{goalachieved invariant probes ustate mstate} \supset \\
&\quad \neg(\text{PCE\_OCCURS probes (invariant ustate) te})
\end{aligned}$$

This is a generic correctness theorem that is independent of the task or any particular machine. It states that under the assumptions that link the machine invariant to the user interaction invariant and the user goal to the machine goal action, the machine specific design rule is “correct”. By correct in this context we mean that if any device whose behaviour satisfies the device specification is used as part of an interactive system with a user behaving according to the principles of cognition as formalised, then no post-completion errors will be made. This is despite the fact that the principles of cognition themselves do not exclude the possibility of post-completion errors.

5 Integration with Full System Verification

Our aim has been to verify a usability design rule in a way that integrates with formal hardware verification. The verification of the design rule needs to consider user behaviour. However, hardware designers and verifiers do not want to be concerned with cognitive models. Our aim has been therefore to separate these distinct interests so that they can be dealt with independently, but within a common framework.

There are several ways the design rule correctness theorem could be used. The most lightweight is to treat the verification of the design rule as a justification of its use in a variety of situations with no further formal reasoning, just an informal argument that any particular device design does match the design rule as specified. Its formal statement then would give a precise statement, including assumptions in the theorem, of what was meant by the design rule. Slightly more formally, the formal statement of the design rule could be instantiated with the details of a particular device. This would give a precise statement about that device. The instantiated design rule correctness theorem then is a specific statement about the absence of user error.

Instantiation involves specifying a user and machine state with entries for each action, the user’s goal, interaction invariant, etc. For example, for a vending machine, the goal might simply be specified as `UserHasChoc`, an accessor to the first entry in the user state, say. The goal event from the machine perspective would be a machine state accessor `GiveChoc`. A further part of the instantiation would be to specify that the invariant was that the value of the user’s possessions (money and chocolate) was at least as high as at the start. The number, and value of each possession is recorded in the user state. A relation `POSS_VAL` calculates the total value. If `possessions` is an accessor function into `ustate`, the invariant for a vending machine is then

$$(\text{POSS\_VAL possessions ustate } t \geq \text{POSS\_VAL possessions ustate } 1)$$

Taking this approach, the final instantiated design rule theorem refers to the specific goals, actions and invariant of the case in point.

A more heavyweight use of the design rule correctness theorem would be to formally verify that the device specification of interest implies such an instantiated design rule. Suppose the device specification for a given vending machine is `VENDING_SPEC mstate`, the goal is given by `GiveChoc` and machine-based invariant by `VND_MINV` then we would prove a theorem of the form:

$$\text{VENDING\_SPEC } mstate \supset \\ \text{MACHINE\_PCE\_DR } (\text{GiveChoc } mstate) (\text{VND\_MINV } mstate) \text{ te}$$

This theorem and its proof only needs to refer to the device specification not the user specification precisely because of the use of a machine-centred version of the design rule. It is independent of the user model and user state.

This theorem can be trivially combined with the design rule correctness statement. This gives a formal result not just that the specification meets the design rule but that in interacting with it a user would not make post-completion errors. For example, if `VND_INV` is the user-centred version of the invariant, `HasChoc` the user-centred version of the goal and `Prbs` accesses the probes from the user state we get an instantiated theorem:

$$(\forall t. \text{VND\_MINV } mstate \ t \supset \text{VND\_INV } ustate \ t) \wedge \\ (\forall t. (\text{HasChoc } ustate \ t) \supset \exists t2. t2 \leq t \wedge (\text{GiveChoc } mstate \ t2)) \wedge \\ \text{USER} \dots (\text{HasChoc } ustate) (\text{VND\_INV } ustate) (\text{Prbs } ustate) \text{ ustate } mstate \wedge \\ \text{VENDING\_SPEC } mstate \supset \\ \neg(\text{PCE\_OCCURS } (\text{Prbs } ustate) (\text{VND\_INV } ustate) \text{ te})$$

Ideally the two assumptions linking the two formalisations of the invariant and the two formalisations of the goal would be discharged. This is the only part of the proof that requires reasoning about the user model. We have isolated it from the verification of the specification meeting its requirements. We obtain a theorem that the user model, using a vending machine that meets the specification, will not make post-completion errors.

$$\text{USER} \dots (\text{HasChoc } ustate) (\text{VND\_INV } ustate) (\text{Prbs } ustate) \text{ ustate } mstate \wedge \\ \text{VENDING\_SPEC } mstate \supset \\ \neg(\text{PCE\_OCCURS } (\text{Prbs } ustate) (\text{VND\_INV } ustate) \text{ te})$$

As the verification framework we have used was originally developed for hardware verification, it would then be simple to combine this result with a hardware verification result stating that the implementation of the device implied its behavioural specification. Suppose we had proved the hardware verification result:

$$\forall mstate. \text{VENDING\_IMPL } mstate \supset \text{VENDING\_SPEC } mstate$$

where `VENDING_IMPL` is a structural specification giving an implementation of the vending machine. We obtain immediately a theorem stating that the *implementation* of the vending machine does not lead to post-completion errors occurring:

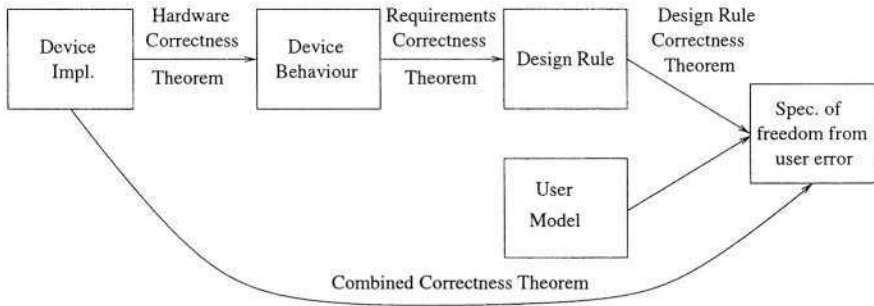


Fig. 3. Combining separate system correctness statements

$$\text{USER} \dots (\text{HasChoc } \text{ustate}) (\text{VND\_INV } \text{ustate}) (\text{Prbs } \text{ustate}) \text{ustate } \text{mstate} \wedge$$

$$\text{VENDING\_IMPL } \text{mstate} \supset$$

$$\neg(\text{PCE\_OCCURS } (\text{Prbs } \text{ustate}) (\text{VND\_INV } \text{ustate}) \text{te})$$

The design rule correctness theorem can thus be combined with a result that a particular device specification meets the design rule. By further combining it with a result that a particular implementation of the device meets the specification we obtain a theorem that the *implementation* does not result in post-completion errors occurring as is illustrated in Figure 3.

The hardware verification is done independently of the cognitive model and explicit usability concerns but then combined with theorems that use them. In previous work [10] [26] we demonstrated how hardware verification correctness theorems could be similarly chained with a full usability task completion correctness theorem stating that when the cognitive model was placed with the behavioural specification of the device, the combined behaviour of the resulting system was such that the task was guaranteed to be completed. The difference here is that the end usability statement being chained to is about the absence of a class of errors rather than task completion; however, the general approach is similar.

6 Conclusions

We have shown how a usability design rule can be verified and the result combined with analysis of other aspects of a design. We started by outlining a set of principles of cognition specifying cognitively plausible behaviour. These principles are based on results from the cognitive science and human-computer interaction literature. From these principles we developed a formal cognitive architecture. This architecture does not directly model erroneous behaviour. Erroneous behaviour emerges if it is placed in an environment (i.e. with a computer system) that allows it.

We then formally specified a class of errors known as post-completion errors. We also specified two versions of a design rule claimed to prevent post-completion

errors. The first is specified in terms of user goals and invariant. The second is in terms of machine events, and so of more direct use to a designer. We proved a theorem that the user-centred design rule is sufficient to prevent the cognitive architecture from committing post-completion errors. This theorem is used to derive a theorem that the machine-based formulation is also sufficient. The resulting theorem is a correctness theorem justifying the design rule. It says that users behaving according to the principles of cognition will not make post-completion errors interacting with a device that satisfies the design rule.

The definitions and theorems are generic and do not commit to any specific task or machine. They are a justification of the design rule in general rather than in any specific case. They can be instantiated to obtain theorems about specific scenarios and then further with specific computer systems.

This work demonstrates an approach that integrates machine-centred verification (hardware verification) with user-centred verification (that user errors are eliminated). The higher-order logic framework adopted is that developed for hardware verification. Specifications, whether of implementations, behaviours or design rules, are higher-order logic relations over signals specifying input or output traces. The theorems developed therefore integrate directly with hardware verification theorems about the computer component of the system. The user based parts of the proof have been isolated from the machine based parts. The theorem developed here, once instantiated for a particular device can be combined with correctness theorems about that device to obtain a theorem stating that the machine implementation implies that no post-completion errors can occur. This requires a proof of a linking theorem that the device specification satisfied the machine-centred design rule.

The work presented here builds on our previous work on fully formal proofs that an interactive system completes a task [11]. A problem with that approach is that with complex systems, guarantees of task completion may be unobtainable. The current approach allows the most important errors for a given application to be focussed on.

7 Further Work

We have only considered one class of error and a simple design rule that prevents it occurring. In doing so we have shown the feasibility of the approach. There are many other classes of error. Others that are potential consequences of our principles of cognition are discussed in [12]. Further work is needed to formally model those error classes and design rules, and verify them formally following the approach developed here. This will also allow us to reason about the scope of different design rules especially those that apparently contradict.

In this paper we have been concerned with the verification of design rules in general, rather than their use in specific cases. We have argued, however that, since the framework used is that developed for hardware verification, integration of instantiated versions of the design rule correctness theorem is straightforward. Major case studies are needed to demonstrate the utility of this approach.

Our architecture is intended to demonstrate the principles of the approach and covers only a small subset of cognitively plausible behaviour. As we develop it, it will give a more accurate description of what is cognitively plausible. We intend to extend it in a variety of ways. As this is done, more erroneous behaviour will be possible. We have essentially made predictions about the effects of following design rules. In broad scope these are well known and based on usability experiments. However, one of our arguments is that more detailed predictions can be made about the scope of the design rules. The predictions resulting from the model could be used as the basis for designing further experiments to validate the model and the correctness theorems proved, or further refine it. We also suggested there are tasks where it might be impossible to produce a design that satisfies all the underlying principles, so that some may need to be sacrificed in particular situations. We intend to explore this issue further.

Acknowledgements. We are grateful to Kirsten Winter and the anonymous referees whose comments have helped greatly improve this paper.

References

1. R. Back, A. Mikhajlova, and J. von Wright. Modeling component environments and interactive programs using iterative choice. Technical Report 200, Turku Centre for Computer Science, sep 1998.
2. A. E. Blandford and R.M. Young. The role of communication goals in interaction. In *Adjunct Proceedings of HCI'98*, pages 14–15, 1998.
3. A.E. Blandford, P.J. Barnard, and M.D. Harrison. Using interaction framework to guide the design of interactive systems. *International Journal of Human Computer Studies*, 43:101–130, 1995.
4. A.E. Blandford, R. Butterworth, and P. Curzon. PUMA footprints: linking theory and craftskill in usability evaluation. In *Proc. of Interact*, pages 577–584, 2001.
5. P. Bumbulis, P.S.C. Alencar, D.D. Cowen, and C.J.P. Lucena. Validating properties of component-based graphical user interfaces. In P. Bodart and J. van der Donckt, editors, *Proc. Design, Specification and Verification of Interactive Systems '96*, pages 347–365. Springer, 1996.
6. R. Butterworth, A.E. Blandford, and D. Duke. Using formal models to explore display based usability issues. *Journal of Visual Languages and Computing*, 10:455–479, 1999.
7. R. Butterworth, A.E. Blandford, and D. Duke. Demonstrating the cognitive plausibility of interactive systems. *Formal Aspects of Computing*, 12:237–259, 2000.
8. M. Byrne and S. Bovair. A working memory model of a common procedural error. *Cognitive Science*, 21(1):31–61, 1997.
9. J.C. Campos and M.D. Harrison. Formally verifying interactive systems: a review. In M.D. Harrison and J.C. Torres, editors, *Design, Specification and Verification of Interactive Systems '97*, pages 109–124. Wien : Springer, 1997.
10. P. Curzon and A.E. Blandford. Using a verification system to reason about post-completion errors. Presented at Design, Specification and Verification of Interactive Systems 2000. Available from <http://www.cs.mdx.ac.uk/puma/> as WP31.

11. P. Curzon and A.E. Blandford. Detecting multiple classes of user errors. In Reed Little and Laurence Nigay, editors, *Proceedings of the 8th IFIP Working Conference on Engineering for Human-Computer Interaction (EHCI'01)*, volume 2254 of *Lecture Notes in Computer Science*, pages 57–71. Springer-Verlag, 2001.
12. P. Curzon and A.E. Blandford. From a formal user model to design rules. In P. Forbrig, B. Urban, J. Vanderdonckt, and Q. Limbourg, editors, *Interactive Systems. Design, Specification and Verification, 9th International Workshop*, volume 2545 of *Lecture Notes in Computer Science*, pages 19–33. Springer, 2002.
13. D.J. Duke, P.J. Barnard, D.A. Duce, and J. May. Syndetic modelling. *Human-Computer Interaction*, 13(4):337–394, 1998.
14. R.E. Fields. Analysis of erroneous actions in the design of critical systems. Technical Report YCST 20001/09, University of York, Department of Computer Science, 2001. D.Phil Thesis.
15. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
16. W. Gray, R.M. Young, and S. Kirschenbaum. Introduction to this special issue on cognitive architectures and human-computer interaction. *Human-Computer Interaction*, 12:301–309, 1997.
17. W.D. Gray. The nature and processing of errors in interactive behavior. *Cognitive Science*, 24(2):205–248, 2000.
18. E. Hollnagel. *Cognitive Reliability and Error Analysis Method*. Elsevier, 1998.
19. D.E. Kieras, S.D. Wood, and D.E. Meyer. Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. *ACM Trans. Computer-Human Interaction*, 4(3):230–275, 1997.
20. D. Leadbetter, P. Lindsay, A. Hussey, A. Neal, and M. Humphreys. Towards model based prediction of human error rates in interactive systems. In *Australian Comp. Sci. Communications: Australasian User Interface Conf.*, volume 23(5), pages 42–49, 2001.
21. T.G. Moher and V. Dirda. Revising mental models to accommodate expectation failures in human-computer dialogues. In *Design, Specification and Verification of Interactive Systems '95*, pages 76–92. Wien : Springer, 1995.
22. F. Paterno' and M. Mezzanotte. Formal analysis of user and system interactions in the CERD case study. In *Proceedings of EHCI'95: IFIP Working Conference on Engineering for Human-Computer Interaction*, pages 213–226. Chapman and Hall Publisher, 1995.
23. F.E. Ritter and R.M. Young. Embodied models as simulated users: introduction to this special issue on using cognitive models to improve interface design. *Int. J. Human-Computer Studies*, 55:1–14, 2001.
24. C. R. Roast. Modelling unwarranted commitment in information artifacts. In S. Chatty and P. Dewan, editors, *Engineering for Human-Computer Interaction*, pages 77–90. Kluwer Academic Press, 1998.
25. J. Rushby. Using model checking to help discover mode confusions and other automation surprises. In *3rd Workshop on Human Error, Safety and System Development (HESSD'99)*, 1999.
26. H. Xiong, P. Curzon, S. Tahar, and A. Blandford. Formally linking MDG and HOL based on a verified MDG system. In M. Butler, L. Petre, and K. Sere, editors, *Proc. of the 3rd International Conference on Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 205–224, 2002.

Using UML Sequence Diagrams as the Basis for a Formal Test Description Language*

Simon Pickin¹ and Jean-Marc Jézéquel²

¹ Dpto. de Ingeniería Telemática, Universidad Carlos III de Madrid, Spain

Simon.Pickin@it.uc3m.es

² IRISA, Campus de Beaulieu, Université de Rennes, France

Abstract. A formal, yet user-friendly, test description language could increase the possibilities for automation in the testing phase while at the same time gaining widespread acceptance. Scenario languages are currently one of the most popular formats for describing interactions between possibly distributed components. The question of giving a solid formal basis to scenario languages such as MSC has also received a lot of attention. In this article, we discuss using one of the most widely-known scenario languages, UML sequence diagrams, as the basis for a formal test description language for use in the distributed system context.

1 Introduction

Testing is crucial to ensuring software quality and the testing phase absorbs a large proportion of development costs. Despite this fact, testing remains more of a craft than a science. As a result, the productivity gains to be obtained from a more systematic treatment of testing, and from the consequent greater level of automation in the testing phase, are potentially very large. The use of a formal test description language is a key part of such a systematic treatment, as explained in [24].

One of the main benefits of such a language is the ability to abstract away from the less important detail. As well as being crucial to managing complexity, abstraction is the means by which platform-independent descriptions are made possible. Graphical analysis and design languages are at the heart of recent moves to raise the level of abstraction in usual software development practice. However, the lack of a formal basis to the most widely-used of these languages, such as UML, limits the extent to which they can be used to facilitate automation. Among such graphical languages, scenario languages are becoming popular for describing interactions between possibly-distributed components, due to the fact that they present the communications and the temporal orderings between them in a clear and intuitive fashion. They would therefore seem to be the notation of choice for describing tests in which the communication aspect is predominant.

* This work was initiated in the COTE project of the French RNTL research programme during Simon Pickin's stay at IRISA

In this article we discuss defining a formal test description language based on UML sequence diagrams and give an overview of such a language called *TeLa*, originally developed in the COTE project [12] (an early version of *TeLa* is presented in [20]). In so doing, we deal with the main semantic issues involved in using UML sequence diagrams, and scenario languages in general, as the basis for a formal language, in our case, for a test description language. Despite the importance of these issues, given the rapid uptake of UML, there is a lack of detailed analyses of them in the literature, not least in the official UML documentation. Here we provide such an analysis not only for UML 1.4/1.5 [16] sequence diagrams but also for the sequence diagrams of the upcoming UML 2.0 [17] standard.

We chose to base our language on UML in order to increase the chances of the work having some industrial impact, notably in widely-used development processes and CASE tools. In addition, this choice should make testing more accessible, not only due to the user-friendly syntax but also since: for a System Under Test (SUT) that is an implementation of a model designed in UML, if the test language is also based on UML, the relation between the tests and (part of) the design model is more manifest. In the component testing context, this accessibility also facilitates the use of tests as component documentation. This documentation can be viewed as a type of constructive contract, as an installation aid, as a regression testing aid in case of changes in the implementation of the component or of its environment, etc. *TeLa* was conceived as a test language for component-based applications. The use of *TeLa* as the interface to the Umlaut UML simulator and the TGV test synthesis tool in the COTE project is treated in [21].

We currently restrict our interest to black-box testing, though scenario languages could also be used for some types of grey-box testing. We aim our language at a higher level of abstraction than, for example, TTCN [6] for which a scenario-based graphical syntax has also been developed in recent years. The UML Test Profile, see [18], is testimony to the industrial interest in a UML-based test description language. The work on *TeLa* reported on here begun before that on the Test Profile and though the two approaches have similarities they are not currently compatible, see Section 3.4.

In Section 2 and Section 3 we analyse the suitability of UML 1.4/1.5 sequence diagrams and UML 2.0 sequence diagrams, respectively. In Section 4 we give a flavour of the *TeLa* language and in Section 5 we present some important aspects of its semantics. In Section 6 we draw conclusions from this work.

2 Suitability of UML 1.4/1.5 Sequence Diagrams

In this section we tackle the issue of basing a formal test description language on UML 1.4/1.5 sequence diagrams, as they are defined in the UML standard. The inconsistencies discussed in Section 2.1 oblige us to modify the semantics as discussed in Section 2.2. The expressiveness of the language must also be increased as discussed in Section 2.3.

2.1 Semantic Inconsistencies of UML 1.4/1.5 Sequence Diagrams

In the UML 1.4/1.5 specifications, the semantics of sequence diagrams is defined in terms of two relations between messages, *predecessor* and *activator*, in a manner similar to [13]. Predecessor relates a message sent by a method to earlier messages sent by the same method while activator relates a message sent by a method to the message that invoked that method. Two messages can then be said to be on the same causal flow if this pair of messages is in the transitive closure of the union of these two relations. Clearly, messages can only be ordered if they are on the same causal flow.

UML 1.4/1.5 sequence diagrams betray their origins, which lie in procedural diagrams—diagrams involving a single causal flow and in which all calls are synchronous—used to describe the behaviour of centralised OO programmes. They represent an attempt to generalise these procedural diagrams to incorporate treatment of concurrency, asynchronous calls, active objects, etc. Unfortunately, the resulting generalisation is not consistent.

In their current state, then, UML 1.4/1.5 sequence diagrams are unsuitable for use as the basis for a formal test description language pitched at a relatively high-level of abstraction. In the following sections we present the main problems with the UML 1.4/1.5 semantics.

The sequence numbering notation. This notation is inconsistent in the following sense. Concurrency is supposed to be modelled using the thread names and explicit predecessors part of the notation. However, this part of the notation appears to assume that messages emitted on *different* lifelines can be related via the predecessor relation, e.g. see Fig. 3-71 of [16], contradicting the semantics (in particular, the semantics of the activation part of the sequence numbering notation that uses the dot notation!).

Moreover, the activation part of this notation is unworkable except in complete causal flows. The use of guards, guarded loops and branching would make it even more unworkable.

Focus bars and asynchronous messages / active objects. It is not clear from the UML standard whether the use of focus bars (or, equivalently, nested sequence numbers) is allowed or desirable in the presence of asynchronous messages and/or active objects. However, if focus bars are not used, according to the semantics, no ordering relations can be inferred between asynchronous messages emitted on different lifelines.

There is little in the standard to justify other interpretations such as that used, without explanation, in [5].

Lack of ordering on incomplete causal flows. Message ordering cannot be specified without complete causal flows. Sequence diagrams cannot therefore be used for specification, where abstraction is fundamental, but only for representing complete execution traces.

Lack of ordering between different causal flows. Even when representing a complete execution trace, messages on different causal flows are not ordered w.r.t. each other.

In particular, a signal is a causal sink that is only related to messages that precede it on the same causal flow. Thus, it has no temporal relation to any message emitted or received on the receiving lifeline except causal ancestors.

Similarly, a message emitted spontaneously by an active object is a causal source that is only related to messages that succeed it on the same causal flow (and even this, only if focus bars / nested sequence numbers are used, see above). Thus, it has no temporal relation to any message emitted or received on the emitting lifeline except causal descendants.

The notion of a message being “completed”. The predecessor relation is said to relate “completed” messages on a lifeline. In the case of asynchronous messages, how does the sender know when a message has “completed”?

The definition of active / passive object. The vagueness of the definition of control flow scheme in UML, i.e. the concept of active/passive object, means that the exact role of this concept in sequence diagrams is not clear.

2.2 Clarifying the Semantics of Sequence Diagrams

In this section we modify the semantics of sequence diagrams to solve the problems discussed in Section 2.1.

An Interworking-Style or “Message-Based” Semantics. The most obvious solution to the problems discussed in the previous section is to remove the activator relation from the semantics and simply relate messages emitted on the same or on different lifelines via the predecessor relation¹. The semantics is therefore defined as a partial order of messages.

There are several ways in which predecessor relations between messages could be inferred from a sequence diagram. The most intuitive solution is to use a semantics similar to that of interworkings [14], but which also allows the use of synchronous calls and focus bars. Note that such a semantics cannot be related to the UML 1.4/1.5 metamodel (i.e. the abstract syntax), since the addition of loops and branching (see below) could lead to infinite depth, and even infinite width, metamodel structures.

An MSC-style or “Event-Based” Semantics. However, an interworking-style semantics is not well-adapted to distributed system description. Neither is it well-adapted to test description since, in the case of messages sent (resp. received) by the tester to (resp. from) the SUT, the tester behaviour to be

¹ In the absence of an activator relation, the question arises as to the role of the focus bar, apart from relating synchronous invocation messages to the corresponding reply messages. We return to this point in Section 2

described encompasses only the emission (resp. reception) of the message by the tester but not its reception (resp. emission) by the SUT.

A semantics which distinguishes emission and reception events is clearly more suitable for describing tester behaviour. We are therefore led to use a partial order of events semantics, similar to that of MSC [10] rather than a partial order of messages semantics, similar to that of interworkings.

A Message-Based Semantics Inside an Event-Based Semantics. In order to be able to use the simplicity of the interworking-style semantics when appropriate, we define a message-based semantics as a restriction of an event-based semantics, extending the work of [4], see [19] for details. This enables us to specify when required that any diagram, or even part of a diagram, that satisfies the RSC (Realisable with Synchronous Communication) property [2], is to be interpreted using the message-based semantics.

Among the multiple uses of this facility, when modelling centralised implementations it can be used to avoid cluttering up diagrams with synchronization messages. In the COTE project, this facility was used to represent the output of the TGV test synthesis tool in sequence-diagram form, see [21].

2.3 Increasing the Expressiveness of Sequence Diagrams

As well as being ill-defined, UML 1.4/1.5 sequence diagrams are lacking constructs which are essential for a test description language of the type discussed in the introduction. Though most of these constructs could also be seen as crucial to other uses of sequence diagrams, here we do not address this wider issue but concentrate, instead, on test description. In looking for suitable constructs, we seek inspiration from the MSC standard. In the following sections we discuss both the constructs we will need to add and the existing constructs which we will need to modify. However, before doing so, we briefly discuss the significant limitations that we do not address.

Perhaps the most important of these limitations is the absence of a gate construct and of a parallel operator, unlike the situation in MSC. This is due to the complexity of their interaction with the loop operator, which would make it all too easy to produce unimplementable sequence diagrams. On the downside, the lack of a parallel operator may make it difficult to describe certain types of distributed testing scenarios involving concurrent choices. Two important limitations which are shared with MSC, namely the absence of a multi-cast construct and the inability to specify the creation of an arbitrary number of components, also deserve a mention.

Sequential Composition. Representation of large numbers of message exchanges requires a means of composing diagrams sequentially. Sequential composition is also essential for representing different alternative continuations, see the section on branching, below.

Use of the event-based semantics means that *weak sequential composition*—that used in the MSC standard—is essential for compositionality, that is, in order

for behaviour to be conserved if a diagram is split into a sequence of smaller diagrams. Though weak sequential composition is the most suitable default composition mechanism, we also require a way of explicitly specifying *strong sequential composition*, that is, composition in which all events of the first diagram precede all events of the second diagram, in order to describe the sequential execution of different test cases. This is needed to model test case termination which involves a global synchronisation in order for a global verdict to be reached. This latter type of sequential composition has no counterpart in MSC.

Internal Action. Modelling component creation/destruction in the presence of lifeline decomposition requires a construct for specifying the creation of a subcomponent on a lifeline. Representing data manipulation requires a construct for specifying assertions and assignments on a lifeline. These three types of action could be placed in an internal action located at a precise position on a lifeline in a similar way to the MSC internal action. Semantically, lifeline termination is also viewed as an internal action. Finally, an “escape” internal action, allowing other language code to be inserted (subject to hypotheses on its effects) is highly desirable in the UML context.

Unlike in MSCs, for more flexibility when used in conjunction with lifeline decomposition, we allow assignment and assertion internal actions to straddle several lifelines, thus involving implicit synchronisations.

Branching. We require a construct to specify the situation in which several alternatives are possible.

If the tester has several possible alternative emission actions, we would normally expect the conditions determining which action is chosen to be fully specified in the test description. However, in the case where the SUT has several possible alternative emission actions, in black-box testing, the conditions determining which action is chosen may depend on details of the internal state of the SUT that are unknown to the tester, particularly if the SUT is a concurrent and/or distributed system. This latter phenomenon is sometimes referred to as *observable non-determinism*. We require a choice construct that is sufficiently general to be able to describe *indefinite choices*, i.e choices involving observable non-determinism.

The “presentation option” branching construct of UML 1.4/1.5 sequence diagrams is unsuitable for the following reasons:

- indefinite choices cannot be specified,
- since the different alternatives appear on the same diagram, diagrams involving branching can only be guaranteed to be unambiguous if causal flows are completely specified,
- if guards are not mutually exclusive, the same behaviour may describe choice or concurrency depending on data values; with any non-trivial data language, therefore, the question of whether a given construct always denotes a choice will often be undecidable.

These properties make this construct of little use for specification. A construct similar to the alternatives of MSCs would answer our requirements.

Loops. We require a construct for specifying general iterative behaviour.

The recurrence construct of UML 1.4/1.5 sequence diagrams is unsuitable since it was conceived for use with lifelines representing multi-objects, without the meaning of emissions and receptions of other messages on such lifelines being clarified. The “presentation option” iteration construct of UML 1.4/1.5 sequence diagrams is unsuitable since it can only be used to specify a finite number of iterations. A construct similar to the MSC loop would answer our requirements.

Explicit Concurrency (Coregion). We require a construct to explicitly break the default ordering on lifelines, e.g. to specify a multi-cast performed by an SUT component, where the tester neither knows, nor cares, about the order in which the messages are emitted.

The explicit predecessor part of the UML sequence numbering notation is unsuitable due to its ambiguity in the presence of loops and its user-unfriendliness. A construct similar to the MSC coregion would answer our requirements. However, we will be led to use a coregion construct that is richer than the MSC coregion. In MSC, focus bars have no semantics. An important part of the semantics we give to focus bars below concerns their interaction with coregions.

Synchronisation Messages / Local Orderings. We require a means of explicitly ordering two events which would not otherwise be ordered. Where the two events are on the same lifeline, this can be used in conjunction with the coregion to specify more complex concurrent behaviour.

In MSC, the general ordering construct is used for this purpose. However, we would prefer to syntactically distinguish general orderings between events on the same lifeline and those between events on different lifelines. While the MSC syntax is adequate for the former, we prefer to use a “virtual synchronisation” message for the latter²

Semantics of Focus Bars. In UML 1.4/1.5 sequence diagrams, focus bars are used to represent method executions. This concept is a valuable one in the context of object- and component-based applications. We would therefore like to give focus bars a semantics consistent with this interpretation in the absence of an activator relation. Focus bars have been introduced in MSCs but have no formal semantics. We consider this situation rather unsatisfactory.

As stated above, focus bars relate request to reply in synchronous invocations. Since we will not use the UML 1.4/1.5 sequence-diagram semantics, we can define the use of focus bars to be optional with asynchronous invocations without losing ordering between events on different lifelines.

We propose to formalise the semantics of the interaction of focus bars with coregions as follows. If a focus bar falls within the scope of a coregion, the ordering relations between the events in the scope of the focus bar are unaffected by that coregion. Thus, a focus bar falling inside a coregion is a shorthand for a set of local orderings, see Fig. 1 for an example.

² It is “virtual” since it denotes new relations but no new events.

In addition, however, we propose that passiveness of the lifeline in question imposes a constraint on the ordering relations between the events in the focus bar scope, on the one hand, and the other events of the coregion that are not in the focus bar scope, on the other. For example, in Fig. 1, Tester 2 being passive corresponds to the constraint that neither of the events $!m_1$ and $!m_8$ can occur between $?m_2$ and $!r_2$ or between $?m_5$ and $!m_7$, where $?$ (resp. $!$) signifies reception (resp. emission). This dependence of the ordering properties on the control flow scheme models the implementation of concurrency on passive components as being via task scheduling rather than via execution threads.

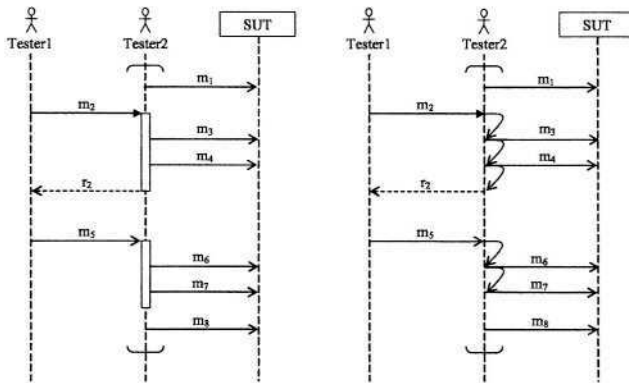


Fig. 1. Focus bars falling inside the scope of a coregion (l.h.s.); equivalent orderings denoted using the local ordering construct (r.h.s.).

Care must be taken as regards the meaning of focus bars in the presence of lifeline decomposition, particularly if the lifeline is specified to be passive. In this latter case, it turns out that we will need to oblige the conservation of focus bars on lifeline composition by using auto-inocations.

Semantics of the Control Flow Scheme Notion. The concept of activeness or passiveness would appear to be a valuable one in object- and component-based applications, albeit a difficult one to pin down. In MSC, this notion does not exist. Again, we consider this situation rather unsatisfactory.

We propose to formalise the notion of passiveness in the sequence-diagram context as a set of constraints on the allowed traces, see [19] for details. If a non-interleaving semantics is used, the notion of passiveness can be defined as a restriction on the allowed linearisations, thus as an (optional) additional semantic layer. These constraints will affect the implementation of concurrency, as described above, and the blocking, or otherwise, of the sender in synchronous calls. Such a definition also helps to ensure that the semantics of the focus bar is consistent with its interpretation as a method execution.

Suspension Region. The suspension region of MSC was introduced to model the blocking of the client during a synchronous invocation. However, we require such blocking to depend on whether or not the lifeline in question represents a passive entity. We have deliberately defined a notion of passiveness that is independent of the basic semantics in order to be able to consider the behaviour of a diagram with, or without, control flow scheme attributes. For this reason, we do not want it to be reflected in the syntax and do not therefore want to use a suspension region construct. Though we do not use such a construct, the vertical region between a synchronous invocation emission event and the reception event of the corresponding synchronous reply has similar behaviour to the focus bar, e.g. concerning interaction with coregions (recall that there may be callbacks) and is similarly affected by the control flow scheme of the lifeline. Below, we refer to this vertical region as the “potential suspension region”.

Symbolic Treatment of Data. The importance of allowing parameters of whole test cases, together with the fact that the values returned by the SUT are unknown, makes a semantics involving symbolic treatment of data inevitable. However, due to the complexity of the task, though the main aspects of the non-interleaving, enumerated-data case and the interleaving, symbolic-data case were formalised in [19], the formalisation of the non-interleaving, symbolic-data case was left for future work.

3 Suitability of UML 2.0 Sequence Diagrams

In this section we discuss the sequence diagrams of the upcoming UML 2.0 standard w.r.t. the requirements presented in the previous section.

UML 2.0 sequence diagrams are heavily inspired by the MSC standard. Their semantics is an MSC-style event-based semantics, thus avoiding many of the problems discussed in Section 2.1. Moreover, most of the additional constructs described in Section 2.3 are present, in particular, weak sequential composition, branching, loops, coregions (a derived operator defined in terms of a parallel operator) and general orderings. Strong sequential composition can be modelled, albeit in a rather cumbersome manner, using the *strict* operator³.

3.1 Some Particular Points of Interest

Some of the main points of interest in attempting to use UML 2.0 sequence diagrams as a basis for a test description language are as follows:

Execution Occurrence and Internal Actions. The execution occurrence construct of UML 2.0 sequence diagrams, denoting a pair of “event occurrences”:

³ In the testing context, if SUT entities are represented explicitly using lifelines, the operands of this operator must not include these SUT lifelines so as not to contradict the black-box hypothesis.

the start and finish of an “execution”, aims at generalising the UML 1.4/1.5 notion of focus bars.

From the definition of the coregion operator of UML 2.0 sequence diagrams, if an execution occurrence (or, in fact, any other “interaction fragment”) falls in the scope of a coregion, the ordering relations between the events in the scope of the execution occurrence are unaffected by that coregion. Thus the situation is similar to that which we propose for focus bars above. However, no interaction with the control flow scheme notion is discussed. In fact, the control flow scheme is not discussed at all in the context of UML 2.0 interactions⁴.

MSC-style internal actions are not present in UML 2.0 sequence diagrams and, since “event occurrences” are either message emissions or receptions, an internal action cannot be modelled by an execution occurrence whose start and finish events are simultaneous, even if such execution occurrences were allowed.

Suspension Region. In [17], the notion of suspension region that was present in earlier drafts has been removed, though there is no indication in the document that this has been done for the same reasons as in our work. Moreover, it is not stated that the “potential suspension region” as defined above constitutes an implicit execution occurrence. Thus, such a region falling inside a coregion may give rise to ambiguities.

Sequence Expression Notation. The sequence numbering scheme remains, in spite of its user-unfriendliness and the fact that it is unworkable both for incomplete causal flows and in the presence of loops, branching and guards. However, in [17], unlike in [16], (thankfully!) its use would seem to be to be restricted to the so-called communication diagrams, which are therefore likely to be of little use outside of the procedural case.

Scope of “Interaction Fragments”. The scope of the “interaction fragment” is not constrained at all in [17]. It is therefore easy to define hard-to-interpret diagrams such as one in which a loop contains the reception of a message but not its emission. The question of defining a set of syntactic restrictions to avoid such problems is currently not tackled.

3.2 Problems with UML 2.0 Constructs Not Present in MSC

The biggest problem with UML 2.0 sequence diagrams concerns the constructs which are new w.r.t MSC, namely the *strict*, *critical region* and *state invariant* constructs and the *neg*, *ignore*, *consider* and *assert* operators.

Though not stated, one assumes that an *ignore* instruction for a message type has priority over a *consider* instruction for the same message type in the surrounding scope and vice versa.

If a valid trace can contain actions which are not in the alphabet of the interaction, is there a need for an *ignore* operator, apart from to cancel the effect

⁴ UML 2.0 sequence diagrams are a concrete syntax for UML 2.0 interactions

of the *consider* or an *assert* operator? If a valid trace cannot contain other such actions, is there a need for an *assert* operator? From Fig. 345 of [17], it would seem that message types not appearing in the interaction can be “considered” using the *consider* operator. If a trace involves the sending and receiving of such a message in the scope of such a *consider* expression is it invalid? If so, why is there a need to use an *assert* operator in Fig. 345? If not, in what way is such a message type to be “considered”?

The new constructs open up a veritable pandora’s box of expressions whose meaning is obscure. For example, what is the meaning of an expression “ignoring” a message type *a* in parallel with an expression involving an occurrence of message type *a*, or with an expression that “considers” message type *a*, or with an expression that asserts the exchange of a message of type *a*? What is the meaning of a *neg* or *ignore* expression in the scope of an *assert* operator? What about an *ignore* or *neg* expression in the scope of a *strict* operator or inside a *critical region*? What is the meaning of a *strict* expression or a *critical region* in parallel with an expression containing some of the same messages?

3.3 UML 2.0 Sequence Diagram Semantics

The semantics of a single UML 2.0 interaction (not that of a pair of such interactions!) is stated in §14.3.7 of [17] to be a set of valid traces and a set of invalid traces. It is also stated that the union of valid traces and invalid traces does not necessarily constitute the “trace universe”, though the exact role of this trace universe in the semantics remains somewhat obscure. It seems reasonable to assume that the (semantic counterparts of the) actions of the interaction are contained in the set of atomic actions used to construct the traces of this trace universe.

The description of the different constructs seems to betray a confusion between, on the one hand, an interaction as a denotation of a set of traces constructed from the (semantic counterparts of the) actions of that interaction—a construction which does not require invoking some mysterious trace universe—and, on the other hand, an interaction as a denotation of a set of traces in some larger trace universe, in the manner of a property.

The interaction-as-property interpretation would work in a manner similar to that in which the test objectives of the TestComposer and Autolink tools [23] are used to select traces from among the set of traces of an SDL specification. It is the description of the *state invariant* construct and the *neg*, *ignore*, *consider* and *assert* operators which reflect the interaction-as-property view. The *ignore*, *consider* and *assert* operators affect how the selection of the traces from the nebulous trace universe is performed

Perhaps both interpretations are intended, that is, an interaction is supposed to denote a set of explicitly-constructed traces which can then be used as a property or selection criteria on any trace universe whose atomic actions contain those of the interaction. Notice that the property operates as both a positive and a negative selection criterion since it selects both a set of valid traces and a set of invalid traces, in a similar way to the accept and reject scenarios of [21].

Even without taking into account the problems with the constructs which are new w.r.t. MSC, it is difficult to judge if the pair-of-trace-sets semantics is viable, since the rules for deriving new pairs of trace sets from combined trace sets are not given. For example, if (a,b) represents a set of valid and (c,d) a set of invalid traces, $\cdot$ denotes sequential composition and $.$ trace concatenation, is it the case that $(a,b) \cdot (c,d) = (a.c, b \cup a.d)$?

In summary, the semantics of UML 2.0 sequence diagrams sketched in [17] is in need of clarification. Moreover, it is far from clear that it could be fleshed out into a consistent semantics for the whole language, i.e. one that includes the constructs that are new w.r.t. MSC. Thus, though our language, TeLa, draws heavily on MSC, it is not completely based on UML 2.0 sequence diagrams.

3.4 The UML Test Profile

The aim of the UML Test Profile (UTP) [18] is similar to that of the language TeLa, but this language is directly based on UML 2.0. Aside from the problems with UML 2.0 sequence diagrams discussed above, it is also the case that the test profile addresses a wider range of issues than our work and has taken a less formal approach than ours. The desire to define a mapping to TTCN-3 and JUnit was of a higher priority than defining a more formal semantics. Furthermore, UTP is less based on sequence diagrams than our approach, often requiring a mix of sequence diagrams and state diagrams.

4 Test Description Language: TeLa

The UML sequence-diagram based language TeLa incorporates the corrections of Section 2.2 and the extra constructs of Section 2.3.

In MSCs, the same behaviour can be modelled either using MSCs with in-line expressions or using HMSCs. Similarly, in TeLa, tests can be described using TeLa one-tier scenario diagrams or TeLa two-tier scenario diagrams. The former comprise TeLa sequence diagrams linked by TeLa sequence diagram references, while the latter comprise TeLa sequence diagrams linked using a TeLa activity diagram. However, in contrast to the situation for MSCs, TeLa one-tier scenario diagrams are less expressive than TeLa two-tier scenario diagrams. This is done with the idea of making them simpler to use and closer to UML 1.4/1.5 syntax as well as with the idea of guaranteeing properties of importance in testing such as that of *concurrent controllability*, see Section 5.3. Examples of one-tier scenario diagrams are given in Figure 2 and Figure 4. The concrete syntax for the choice and loop operator using auto-involutions was chosen in the COTE project to be easy to implement in the Objecteering UML tool; clearly, a better concrete syntax could be devised. The equivalent two-tier scenario diagrams are shown in Figure 3 and Figure 5. Since TeLa two-tier scenario diagrams were developed in the COTE project, similar structures, “interaction overview” diagrams, have been introduced in UML 2.0.

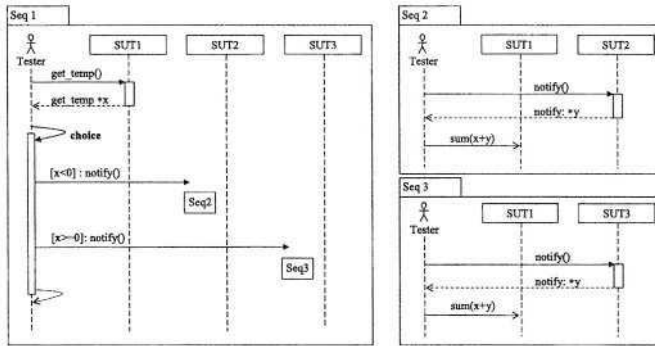


Fig. 2. A TeLa one-tier scenario diagram showing a TeLa sequence-diagram choice and describing the same behaviour as the diagram of Fig. 3.

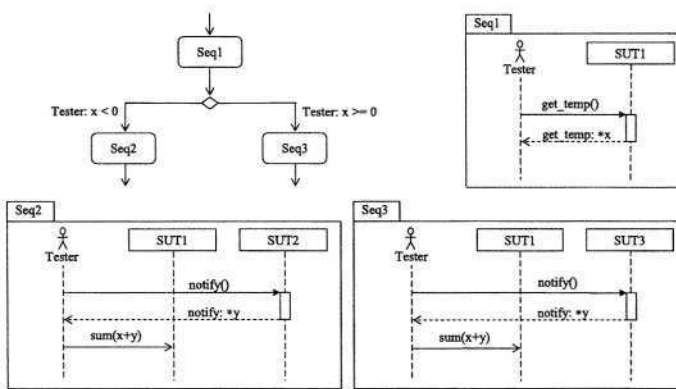


Fig. 3. A TeLa two-tier scenario diagram showing a TeLa activity-diagram choice and describing the same behaviour as the diagram of Fig. 2.

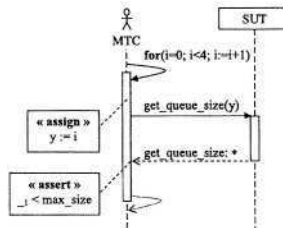


Fig. 4. A TeLa one-tier scenario diagram showing a TeLa sequence-diagram loop and describing the same behaviour as the diagram of Fig. 5.

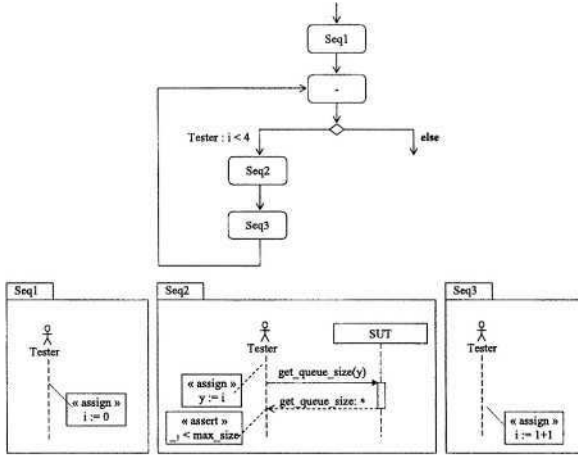


Fig. 5. A TeLa two-tier scenario diagram showing a TeLa activity-diagram loop and describing the same behaviour as the diagram of Fig. 4.

It is worth mentioning that TeLa sequence diagram loops must be restricted to diagrams with the RSC property [2] in order to be able to define their scope via two valid cuts, see [9]. These are the two valid cuts that contain all events occurring in the loop scope on the lifeline on which the loop is defined, together with the minimum number of events located on other lifelines. Moreover, we do not allow TeLa sequence diagram loops to occur inside coregions, in order for every sequence-diagram loop to be equivalent to an activity-diagram loop, without the need for a parallel operator at the activity-diagram level.

5 TeLa Semantics

In this section we tackle the main issues involved in giving a semantics to our test description language. This includes dealing with the question of verdicts, and that of determinism, in order to answer the question of when a test description defines a test case. To date these issues have only been addressed very superficially in the non-interleaving context, e.g. see [15] and [3].

5.1 Structural Semantics

In order to define a framework for lifeline decomposition we define a structural semantics in terms of an underlying hierarchical component model, see [19] for details. By comparison, lifeline decomposition is not addressed in TTCN-3 GFT or UTP. A valid cut of a sequence diagram, see [9], is mapped to a snapshot of this underlying component model (snapshots are necessary due to component creation and destruction). This gives us a means to define the internal structure

of components, from the two default components, the tester and the SUT, down to the base-level components. The base-level components are the owners of the dynamic variables used in the specification. We also introduce the use of a dot notation on arrow labels to identify the target port and (base-level) originating port of a message.

This underlying component model is defined by the test architecture specification and possibly also by the specification of the assumed SUT structure. We propose to use UML 2.0 component diagrams as the concrete syntax for our component specifications, augmenting these diagrams with annotations concerning component properties (see below) and the implemented communication architecture. The underlying component model can be used to give a clear meaning to the use of constructs such as focus bars and internal actions under lifeline decomposition. As already stated, in MSC, focus bars have no semantics and the meaning of internal actions or guards in the presence of lifeline decomposition is not addressed. One assumes they are only allowed on lifelines representing base-level entities.

Finally, the structural semantics also provides a framework for deployment and for defining component properties which affect the interpretation of the diagrams. The property of being message-based or event-based and that of being active or passive are examples of such properties. Note that if a component is msg-based, all its subcomponents are msg-based and if a component is active, at least one of its subcomponents must be active.

5.2 Test Specific Considerations: Representation of SUT

As stated in Section 1, a test description is only required to specify the events on the tester. However, unlike TTCN GFT, we choose to represent the SUT explicitly using one or several lifelines. Recall that we cannot use gates since we have excluded these from the language in order to avoid semantic complexity. However, we, in any case, consider the use of explicit SUT lifelines the most user-friendly representation for the following reasons:

- it is closer to the current standard usage of UML sequence diagrams,
- the relation between a UML model and the tests of an implementation of (part of) that model is clearer,
- the situation in which tester events are ordered via SUT events is better communicated by representing the message exchanges involved explicitly,
- it does not give the impression that a specific communication architecture (e.g. FIFO queues per channel) is being denoted; we prefer this to be specified as part of a separate test architecture diagram.

The semantics is then given in two stages: first, derive a partial order of events including SUT events; second, project this partial order onto the tester events, c.f. projection of MSCs in [7]. A choice that is local in the second stage semantics, a *test local choice*, is not necessarily a local choice, i.e. a choice that is local in the first stage semantics. See Fig. 6 for an example of a local, but test non-local, choice. Semantics via projection can also give rise to additional non-determinism.

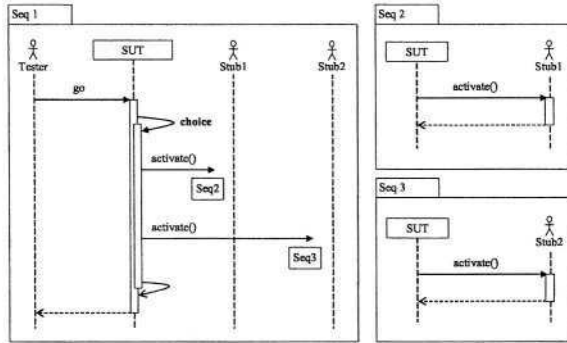


Fig. 6. A TeLa one-tier scenario diagram showing a test non-local choice

Internal Structure. We are now in a position to answer the question of what lifelines represent in TeLa. The tester lifelines represent subcomponents of the single tester component⁵. As we are in a black-box testing context, the SUT lifelines usually represent ports of the SUT component. However, we also allow them to represent the assumed SUT internal structure in terms of subcomponents, if an SUT model is available.

5.3 Dynamic Semantics

We base our semantics on the event-structure semantics defined for MSCs in [8]. According to the classification of [22], this is a non-interleaving, branching-time, behavioural semantics. We choose a non-interleaving semantics to clearly distinguish between choice and concurrency and since it is more appropriate for the distributed system context, in particular, for discussing local and global verdicts. We choose a branching-time semantics since this gives us a framework in which to discuss ideas of determinism and controllability. The set of linearisations of the event structures of [8] defines an interleaving, linear-time, behavioural semantics.

Non-interleaving Input / Output models. Input-output models (I/O models), in which the externally visible actions of a specification are divided into inputs and outputs, have proved to be the most applicable in the testing context, see [1] for a survey. To the inputs, resp. outputs of the tester correspond outputs, resp. inputs of the SUT. Our aim is to generalise the use of interleaving input-output models in describing centralised test cases, see [24], to the use of non-interleaving input-output models in describing distributed test cases. We see this as the first step towards generalising the formal approach to conformance testing as a whole.

⁵ Unlike in [17], here, the terms “component” and “port” denote instances of types rather than types

To deal with internal tester structure and distributed testers, we add the notion of internal action. The tester inputs (emitted by the SUT)—the *observable actions*—are actions for which the test description environment, i.e. the SUT, has the initiative, while the internal actions and tester outputs—the *controllable actions*—are actions for which the tester has the initiative.

Determinism and Controllability in the Non-interleaving Context. In this section, due to the complexity involved, we do not deal with symbolic data and assume that all data is enumerated.

In the input-output model context the automata-theory definition of determinism (no state has multiple identically-labelled output transitions) does not coincide with the “intuitive” notion of determinism (any state with an outgoing transition labelled by an output action has no other outgoing transitions). In this context, the “intuitive” notion of determinism is often termed *controllability*. Both types of determinism are of importance in testing theory. For example, the *test graphs* of [11] are deterministic automata while *test cases* are controllable test graphs. In extending the established input-output testing models to the interleaving case we further refine the two above types of determinism.

In [19], we define the notion of *minimally deterministic test description* as one in which no two concurrent events (resp. events in minimal conflict) are labelled by the same action (resp. observable action). Thus, w.r.t. the usual event-structure definition of determinism, minimal determinism allows minimal conflicts involving events labelled by the same controllable action. However, we also add the condition that any such conflicts must either be resolved on the occurrence of a controllable action or must result in identical verdicts. In terms of the official MSC Semantics, minimal determinism prevents a delayed choice occurring on a fail verdict and therefore ensures that fail verdicts are well-defined. We put forward minimally-deterministic, test descriptions as the non-interleaving analogues of the test graphs of [11].

In [19], we define five notions of controllability ranging from *essential controllability* (EC) through *concurrent controllable* (CC) to *full controllability* (FC). The other two notions are obtained by using the distinction between tester internal actions and tester output actions. A test description is said to be EC if it is minimally deterministic and no event labelled by a controllable action is in minimal conflict with any other event. It is said to be CC if it is EC and no event labelled by a controllable action is concurrent with one labelled by an observable action. It is said to be FC if it is EC and no event labelled by a controllable action is concurrent with any other event.

Defining different types of test case according to whether they have the appropriate controllability property gives us five types of parallel test case. We use the terms *parallel test case*, *coherent parallel test case* and *centralisable test case* for the types corresponding to the above three properties.

Verdicts. Another crucial aspect of test description is modelling verdicts. In TeLa, implicit local verdicts are used as follows:

- if the behaviour completes as shown, the verdict is *pass*
- if an unspecified reception from the SUT is obtained, the verdict is *fail*
- in the non-enumerated data case, if one of a set of concurrent guards of the tester does not evaluate to true, or none of a set of alternative guards of the tester evaluates to true, the (local) verdict is inconclusive.

Fail is an existential notion so a single local fail implies a global fail. Pass is a universal notion, so a local pass on all concurrent branches implies a global pass. The meaning of a local inconclusive verdict is that the component which derives the verdict performs no more actions while the other components continue until termination or deadlock to see if they can derive a fail verdict. Communication of local verdicts to the entity responsible for emitting the global verdict is not explicitly modelled in TeLa.

Implicit verdicts represent a higher level of abstraction than the TTCN-3 defaults. The latter have also been taken up by the UTP proposal. We contend that implicit verdicts are also easier to use. However, in TeLa, we also allow certain types of fail and inconclusive verdicts to be explicitly specified for the situation in which this is more convenient. Use of explicit verdicts requires certain restrictions on the test description in order for them to be well defined, see [19].

The above notion of verdicts is formalised in [19] for the non-interleaving case but without symbolic treatment of data⁶. Here, we briefly sketch the basis of this formalisation.

We first define a verdict as an annotation on terminal events, that is, events with no successors. We say that a configuration is *complete* w.r.t. a set of observable actions if for each action of the set, there is an enabled event of the configuration labelled by that action. We say that an event structure is *test complete* if all configurations having an enabled action are complete and all terminal events are annotated with a verdict.

We define a fail configuration as a configuration that includes a fail event, a pass configuration as a maximal configuration whose terminal events are all pass events and an inconclusive configuration as a configuration that includes an inconclusive event but does not include a fail event. A *maximal test configuration* is a configuration with an associated verdict. An *execution* is a set of configurations that is totally ordered by inclusion, where each element extends its predecessor by a single event. A *maximal test execution* is an execution whose largest element is a maximal test configuration. The *test verdict* of a maximal test execution is the verdict associated to the last element of the execution.

In order for verdicts to be consistently defined, we must impose the condition that isomorphic configurations of a test-complete event structure have identical verdict annotations. This ensures that any two maximal test runs having the same trace have the same associated verdict. If there are no inconclusive verdicts, this can be guaranteed by demanding minimal determinism, and if there are, by demanding determinism.

Concerning the symbolic data case, we briefly mention a point of interest concerning guards and assertions. Normally, if a guard is not satisfied, the exe-

⁶ And also for the non-interleaving, symbolic data case

cution path is not feasible whereas if an assertion is not satisfied, an exception is raised. In the presence of implicit verdicts however, this distinction is blurred since if a guard is not satisfied, a fail verdict, which can be viewed as a kind of exception, may result.

6 Conclusion

We have clarified the problems in using UML sequence diagrams as the basis for a formal test description language and have sketched the solution to these problems and the other main semantic issues, as implemented in TeLa, see [19] for more details. The use of this language in test synthesis is described in [21].

References

- [1] Brinksma, E., Tretmans, J.: Testing transition systems: An annotated bibliography. In: Cassez, F., Jard, C., Rozoy, B., and Ryan, M. (Eds.): *Modelling and Verification of Parallel Processes*. Proc. of Summer School MOVEP'00. (2000).
- [2] Charron-Bost B., Mattern, F., Tel, G.: *Synchronous, Asynchronous and Ordered Communication*. Distributed Computing 9(4). Springer-Verlag (1996).
- [3] Deussen, P.H., Tobies, S.: Formal Test Purposes and the Validity of Test Cases. In: Peled, D. Vardi, M. (Eds.): *Formal Techniques for Networked and Distributed Systems* (Proc. FORTE 2002). Lecture Notes in Computer Science Vol. 2529. Springer-Verlag (2002).
- [4] Engels, A., Mauw, S. Reniers, M.A.: A Hierarchy of Communication Models for Message Sequence Charts. *Science of Computer Programming* 44(3). Elsevier North-Holland (2002).
- [5] European Telecommunications Standards Institute (ETSI): Method for Testing and Specification (MTS); Methodological Approach to the Use of Object-Oriented in the Standards Making Process. ETSI Guide EG 201 872, V1.2.1. ETSI (2001).
- [6] European Telecommunications Standards Institute (ETSI): Method for Testing and Specification (MTS); The Testing and Test Control Notation version 3. ETSI Standard ES 201 873 Parts 1 to 6, V2.2.1. ETSI (2003).
- [7] Genest, B., Hélouët, L., Muscholl, A.: High-Level Message Sequence Charts and Projections. In: Goos, G., Hartmanis, J., van Leeuwen J. (Eds.): *CONCUR 2003 - Concurrency Theory* (Proc. CONCUR 2003). Lecture Notes in Computer Science Vol. 2761. Springer Verlag (2003).
- [8] Hélouët, L., Jard, C., Caillaud, B.,: An Event Structure Based Semantics for Message Sequence Charts. *Mathematical Structures in Computer Science* Vol. 12. Cambridge University Press (2002).
- [9] Hélouët, L., Le Maigat, P.: Decomposition of Message Sequence Charts. Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM 2000). Grenoble, France (2000). See: <http://www.irisa.fr/manifestations/2000/sam2000/papers.html>.
- [10] International Telecommunications Union—Telecommunication Standardization Sector (ITU-T): Message Sequence Chart. Recommendation Z.120. ITU-T (1999).
- [11] Jard, C., Jérón, T.: TGV: Theory, Principles and Algorithms. In: Proc. 6th world conference on Integrated Design and Process Technology (IDPT02). (2002)

- [12] Jard, C., Pickin, S.: COTE—Component Testing Using the Unified Modelling Language. ERCIM News Issue 48. ERCIM EEIG (2001).
- [13] Lamport, L.: On Interprocess Communication. *Distributed Computing* 1(2). Springer Verlag (1986).
- [14] Mauw, S., Wijk van M., Winter, T.: A Formal Semantics of Synchronous Interworkings. In: Faergemand, Sarma, A. (Eds.): *SDL'93—Using Objects* (Proc. SDL Forum 93). Elsevier North-Holland (1993).
- [15] Mitchell, B.: Characterising Concurrent Tests Based on Message Sequence Chart Requirements. In: *Proc. Applied Telecommunication Symposium*. (2001)
- [16] Object Management Group (OMG): Unified Modelling Language Specification version 1.5. OMG, Needham, MA, USA (Mar. 2003).
- [17] Object Management Group (OMG): UML 2.0 Superstructure Specification. OMG, Needham, MA, USA (Aug. 2003).
- [18] Object Management Group (OMG): UML Testing Profile, version 2.0. OMG, Needham, MA, USA (Aug. 2003).
- [19] Pickin, S.: Test des Composants Logiciels pour les Télécommunications. Ph.D. Thesis. Université de Rennes, France (2003).
- [20] Pickin, S., Jard, C., Heuillard, T., Jézéquel, J.M., Defray, P.: A UML-integrated Test Description Language for Component Testing. In: Evans, A., France, R., Moreira, A., Rumpe, B. (Eds.): *Practical UML-Based Rigorous Development Methods. Lecture Notes in Informatics (GI Series), Vol. P7*. Kollen-Druck + Verlag (2001).
- [21] Pickin, S., Jard, C., Le Traon, Y., Jézéquel, J.M., Le Guennec, A.: System Test Synthesis from UML Models of Distributed Software. In: Peled, D. Vardi, M. (Eds.): *Formal Techniques for Networked and Distributed Systems* (Proc. FORTE 2002). *Lecture Notes in Computer Science* Vol. 2529. Springer-Verlag (2002).
- [22] Sassone, A., Nielsen, M., Winskel, G.: Models for Concurrency: Towards a Classification. *Theoretical Computer Science* 170(1–2) Elsevier (1996).
- [23] Schmitt, M., Ebner, M., Grabowski, J.: Test Generation with Autolink and Test Composer. In: *Proc. 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM 2000)*. Grenoble, France (2000). See: <http://www.irisa.fr/manifestations/2000/sam2000/papers.html>.
- [24] Tretmans, J.: Specification Based Testing with Formal Methods: From Theory via Tools to Applications. In: A. Fantechi, A. (Ed.): *FORTE / PSTV 2000 Tutorial Notes*. (2000).

Viewpoint-Based Testing of Concurrent Components

Luke Wildman, Roger Duke, and Paul Strooper

School of Information Technology and Electrical Engineering,
The University of Queensland,
{luke,rduke,pstroop}@itee.uq.edu.au,
Fax: +61 7 3365 4999,
Phone: +61 7 3365 2097

Abstract. The use of multiple partial viewpoints is recommended for specification. We believe they also can be useful for devising strategies for testing. In this paper, we use Object-Z to formally specify concurrent Java components from viewpoints based on the separation of application and synchronisation concerns inherent in Java monitors. We then use the Test-Template Framework on the Object-Z viewpoints to devise a strategy for testing the components. When combining the test templates for the different viewpoints we focus on the observable behaviour of the application to systematically derive a practical testing strategy. The Producer-Consumer and Readers-Writers problems are considered as case studies.

Keywords: Viewpoints, Object-Z, Test Template Framework, Concurrency, Java

1 Introduction

Concurrent programs are notoriously difficult to test because of the ways in which threads can synchronise and interact with each other. In this paper, we focus on testing concurrent Java components and we assume that the component can be accessed by any number of threads. We apply a specification-based testing approach to derive a strategy for testing such concurrent components. When devising a strategy for testing concurrent components, one has to assume a basic concurrency model whether it is the Java monitors model (as in our case) or another (more generic) model. Our starting point for deriving a testing strategy is a formal, Object-Z [7] model of a Java monitor. We apply the Test Template Framework [19] to this model to derive generic test conditions for Java monitors. However, the strategy will be applied to specific components, so we need to consider what happens when we combine this information with application-specific test conditions. To do this, we take a viewpoints-based approach. We combine the generic test conditions from the Java monitor model with test conditions derived from an Object-Z specification of a model of the application we want to test. As we illustrate using a Producer-Consumer monitor,

if we do this naively, then the number of test conditions becomes unmanageable. That is, if we model the Producer-Consumer application in Object-Z, apply the Test Template Framework, and combine the resulting test conditions with the generic ones, we obtain a very large number of test conditions. Moreover, many of these conditions test the generic behaviour of Java monitors, rather than the application-specific behaviour of the Producer-Consumer monitor that we are interested in.

To alleviate this problem, we use a restricted viewpoint of the general model of Java monitors that focuses on those aspects of the model that have an externally visible effect on the behaviour of the concurrent component. By combining the test conditions from this restricted viewpoint with the application-specific test conditions, we reduce the number of test conditions and can focus on those conditions that are relevant to the observable behaviour of the application.

Although we do not discuss it in detail, the test conditions that are generated by the approach described in this paper could be used to generate test sequences in a testing tool such as ConAn [12].

1.1 Related Work

We apply the Test Template Framework [19] to concurrent Java components in Object-Z [7]. Test Templates have been generated from Object-Z before [5] and there has been some work dealing with the issues of concurrency under the guise of interactive systems [15]. Interactive systems have also been considered by others [1]. However, the Test Template Framework has not previously been applied directly to concurrent components.

The use of the Test Template Framework with inheritance has been considered elsewhere [17], as has the combination of Object-Z operations [18]. However, neither deal with the issues of multiple inheritance as we do here.

We use viewpoint-based specifications [10,3]. That is, multiple partial specification are used rather than a single monolithic specification. This approach allows different concerns to be treated separately without constructing a complicated unified model. Viewpoints-based testing has been considered before [14, 4], but in this previous work different viewpoints are represented in different specification languages, whereas we use a single specification language.

1.2 Overview

An introduction to Java and the Java concurrency model is given in Section 2. This includes a formal specification of a Java monitor in Object-Z. In Section 3 the Test Template Framework is described and applied to the Java monitor specification to produce a set of test conditions. The specification of the Producer-Consumer monitor is considered in Section 4 along with test conditions for the application-specific behaviour. In Section 5 a naive combination of the test hierarchies from the two viewpoints is discussed and a different approach based on a restricted concurrency viewpoint is presented. To further evaluate and illustrate

the approach, its application to the Readers-Writers problem is considered in Section 6. Concluding remarks are presented in Section 7.

2 Java Concurrency

A typical application involving Java concurrency is presented in Figure 1. The Java code implements a finite buffer which may be shared between many Producer and Consumer threads for the purpose of communicating resources. We shall assume a basic understanding of the Java synchronisation model. The put

```
public class BufferImpl {

    protected int[] buf;
    protected int in = 0;
    protected int out= 0;
    protected int count= 0;
    protected int size;

    public BufferImpl(int size) {
        this.size = size;
        buf = new int[size];
    }

    public synchronized void put(int i) throws InterruptedException {
        while (count==size) wait();
        buf[in] = i;
        ++count;
        in=(in+1) % size;
        notifyAll();
    }

    public synchronized int get() throws InterruptedException {
        while (count==0) wait();
        int i =buf[out];
        --count;
        out=(out+1) % size;
        notifyAll();
        return (i);
    }
}
```

Fig. 1. A Java implementation of a finite buffer.

method is used to add a resource to the finite buffer. The calling thread waits until space is available and then notifies all waiting threads after it adds the resource. The get method retrieves a resource from the buffer. The calling thread

waits until a resource is available and then notifies all other waiting threads after it removes the resource. An object of the class `BufferImpl` conforms to the underlying Java thread synchronisation model based on synchronised methods and blocks of code, together with the Java methods `wait`, `notify` and `notifyAll` inherited from the Java `Object` superclass. (Other Java features like thread creation, `join`, `sleep` and `interrupt`, or the deprecated `suspend`, `resume` and `stop` will not be discussed.)

In previous work [8], the specification of the class *Object* (presented below) was used to capture the underlying Java concurrency mechanism consistent with that described within the Java Language Specification [9] and the Java Virtual Machine (JVM) Specification [11]. Instances (i.e. objects) of the class *Object* denote the objects in a Java system, such as instances of `BufferImpl`; the class *Object* captures the underlying concurrency of the system from the viewpoint of these objects.

In the specification of *Object*, *Thread* denotes the set of all possible program threads in a Java system. The set *OneThread* is defined by

$$OneThread == \{st : \mathbb{P} Thread \mid \#st \leq 1\}$$

and denotes subsets of threads containing at most one thread.

Consider now the class *Object* in detail. A basic understanding of Object-Z is assumed. The three state attributes each denote subsets of threads. The attribute *isLockedBy* denotes the thread (if any) that holds the lock on the object, *isBlocking* denotes the set of threads blocked on the object, and *hasAskedToWait* denotes the set of threads that are waiting to be notified by the object. These three subsets are mutually disjoint, capturing the requirement that at any given time a thread can play at most one of these roles for any given object. Initially all three subsets are empty.

A call to a synchronised method or entry into a synchronised block is modelled in two steps. The thread initiates a *lockRequestedByThread* and the JVM responds with a *giveLockToThread*. The operation *lockRequestedByThread* captures the object's view of the situation when a Java thread seeks to enter a synchronised block of an object it is not already locking. We do not model recursive locks. (If a thread holds the lock on an object, in our model we assume that it can enter any synchronised block of that object without restriction. In particular, it does not need to request a further lock on the object.) The thread in question cannot be blocked on the object and cannot be waiting to be notified by the object. The outcome of the operation is that the thread joins the set of threads blocked by the object.

The operation *giveLockToThread* specifies what happens when the JVM selects a thread from among those currently blocked on the object and allows that thread to lock it. This captures the object's view of the situation when a Java thread is given access to the synchronised blocks of the object. The object cannot already be locked, and the outcome of the operation is that the thread is given the lock on the object and is removed from the set of blocked threads. *giveLockToThread* can occur whenever the object is not locked, i.e., initially, or whenever a thread releases the lock as described by the following 2 operations.

The operation *lockReleasedByThread* specifies what happens when the thread that currently holds the lock on the object releases that lock. This captures the object's view of the situation when the Java thread currently locking the object no longer requires access to any synchronised block of the object. The outcome is that the object is no longer locked.

Object	INIT
$\begin{array}{l} isLockedBy : OneThread \\ isBlocking : \mathbb{P} Thread \\ hasAskedToWait : \mathbb{P} Thread \\ \hline disjoint(isLockedBy, isBlocking, \\ \quad hasAskedToWait) \end{array}$	$\begin{array}{l} isLockedBy = \emptyset \\ isBlocking = \emptyset \\ hasAskedToWait = \emptyset \end{array}$
$\begin{array}{l} lockRequestedByThread \\ \Delta(isBlocking) \\ t? : Thread \\ \hline t? \notin isLockedBy \\ t? \notin isBlocking \\ t? \notin hasAskedToWait \\ isBlocking' = isBlocking \cup \{t?\} \end{array}$	$\begin{array}{l} giveLockToThread \\ \Delta(isLockedBy, isBlocking) \\ t! : Thread \\ \hline isLockedBy = \emptyset \\ t! \in isBlocking \\ isLockedBy' = \{t!\} \\ isBlocking' = isBlocking \setminus \{t!\} \end{array}$
$\begin{array}{l} lockReleasedByThread \\ \Delta(isLockedBy) \\ t? : Thread \\ \hline isLockedBy = \{t?\} \\ isLockedBy' = \emptyset \end{array}$	$\begin{array}{l} askThreadToWait \\ \Delta(isLockedBy, hasAskedToWait) \\ t! : Thread \\ \hline isLockedBy = \{t!\} \\ isLockedBy' = \emptyset \\ hasAskedToWait' = \\ \quad hasAskedToWait \cup \{t!\} \end{array}$
$\begin{array}{l} notifyThread \\ \Delta(isBlocking, hasAskedToWait) \\ t! : OneThread \\ \hline isLockedBy \neq \emptyset \\ t! \subseteq hasAskedToWait \\ \#hasAskedToWait > 0 \Rightarrow \\ \quad \#t! = 1 \\ hasAskedToWait' = \\ \quad hasAskedToWait \setminus t! \\ isBlocking' = isBlocking \cup t! \end{array}$	$\begin{array}{l} notifyAllThreads \\ \Delta(isBlocking, hasAskedToWait) \\ st! : \mathbb{P} Thread \\ \hline isLockedBy \neq \emptyset \\ st! = hasAskedToWait \\ hasAskedToWait' = \emptyset \\ isBlocking' = isBlocking \cup st! \end{array}$

The operation *askThreadToWait* specifies what happens when the object requests that the thread currently holding the lock on the object wait for notification. This captures the object's view of the situation when a Java thread

executes a wait while accessing a synchronised block of the object. The outcome is that the thread is added to the set of threads waiting to be notified, and the object is no longer locked.

The operation *notifyThread* specifies what happens when the JVM selects a thread from among those currently waiting on the object and notifies it. This captures the object's view of the situation when a notify is executed by some other thread currently accessing a synchronised block of the object. The outcome is that the selected thread is removed from the set of waiting threads and added to the set of threads blocked on the object. Note that *notifyThread* does nothing if no threads are waiting, i.e. in this case execution of notify will wake no threads.

The operation *notifyAllThreads* is like *notifyThread* except that all threads waiting on the object are notified. This captures the object's view of the situation when a notifyAll is executed by some other thread currently accessing a synchronised block of the object. The outcome is that all the threads currently waiting on the object are added to the set of threads blocked on the object, while the set of threads waiting on the object becomes empty.

3 Object Class Test Cases

3.1 Test Template Framework

The Test Template Framework (TTF) [19] provides a systematic way to select and record abstract test cases for individual operations from a model-based formal specification. Test case selection remains under the control of the human tester and multiple testing strategies are supported. The original work focused on specifications written in the Z notation, but the framework has been extended for Object-Z specifications [5].

3.2 Process

The framework provides a systematic way of finding partitions for each operation. The precondition of an operation is used as the starting point for partitioning since Object-Z operations are disabled for inputs outside the precondition. The precondition is called the *valid input space* (VIS). The valid input space is a subset of the *input space* (IS) of an operation, which is defined as the restriction of the operation's signature to input components (inputs and pre-state components). The *output space* (OS) is similarly defined over the output components.

A *strategy* in the framework identifies a particular technique for deriving test cases. Both traditional techniques such as input partitioning and boundary analysis, and specialised techniques that exploit the specification notation are used. The framework encourages the use of multiple strategies to build the *test template hierarchy* (TTH). The hierarchy captures the relationships between test templates and strategies, and serves as a record of the test development process. The root of the hierarchy is the valid input space.

The hierarchy is created by applying testing strategies to existing test templates to derive additional ones. A test template hierarchy is usually a directed

acyclic graph with the leaf nodes partitioning the valid input space. Strategies are applied to the hierarchy until the tester is satisfied that the leaf templates of the hierarchy represent adequate sources of tests, that is, every instantiation of a leaf template is equally likely to reveal an error in an implementation.

To identify test data, the tester instantiates the leaf test templates (TT) in the test template hierarchy by supplying specific values for the state and inputs. The resulting test templates are called instance templates (IT). The framework also defines output templates (OT) corresponding to test or instance templates. These define the expected final states and outputs corresponding to the test or instance template. The output is calculated by restricting the operation to the input described in the template and then projecting onto the operation's output space.

3.3 Object TTH

We now discuss the application of the Test Template Framework to the *Object* specification. We discuss the derivation for the *lockRequestedByThread* operation in detail for each step and mention the interesting aspects of the application to the other operations.

Note that the *INIT* schema is ignored because it is not an operation. That is, it has no input space to which we can apply the Test Template Framework.

Valid Input Space. The Valid Input Space for *lockRequestedByThread* is:

$$VIS_{lockRequestedByThread} \hat{=} \text{pre } lockRequestedByThread$$

which expands and simplifies to

$$VIS_{lockRequestedByThread} \hat{=} \left[\begin{array}{l} isLockedBy : OneThread; isBlocking, hasAskedToWait : \mathbb{P} Thread; \\ t? : Thread \mid \text{disjoint}(isLockedBy, isBlocking, hasAskedToWait) \wedge \\ t? \notin isLockedBy \wedge t? \notin isBlocking \wedge t? \notin hasAskedToWait \end{array} \right]$$

Test Templates. The *lockRequestedByThread* test template hierarchy is defined as follows (taking the set of all possible strategies as given):

$$\begin{array}{l} [Strategies]; \\ TTH_{lockRequestedByThread} : \\ TT_{lockRequestedByThread} \times Strategy \rightarrow \mathbb{P} TT_{lockRequestedByThread} \end{array}$$

The function $TTH_{lockRequestedByThread}$ captures the combination of test templates with testing strategies to derive additional test templates. For most strategies, the children partition the parent.

A common, intuitive testing strategy is 0-1-many, based on the cardinality of a set such as *isBlocking*. This strategy is referred to as (one example of) type-based selection. Type-based selection ($TB : Strategies$) is identified as a

particular testing strategy and is used to partition the valid input space into cases where *isBlocking* is empty, a singleton, and a set with more than one element, generating three test templates (distinguished with numeric subscripts) as a result.

$$\begin{aligned} TT_{lockRequestedByThread.0} &\hat{=} \left\{ VIS_{lockRequestedByThread} \mid \#isBlocking = 0 \right\} \\ TT_{lockRequestedByThread.1} &\hat{=} \left\{ VIS_{lockRequestedByThread} \mid \#isBlocking = 1 \right\} \\ TT_{lockRequestedByThread.2} &\hat{=} \left\{ VIS_{lockRequestedByThread} \mid \#isBlocking > 1 \right\} \end{aligned}$$

$$TTH_{lockRequestedByThread}(VIS_{lockRequestedByThread}, TB) = \{ TT_{lockRequestedByThread.0}, TT_{lockRequestedByThread.1}, TT_{lockRequestedByThread.2} \}$$

We can apply a similar type-based selection strategy to both the *isLockedBy* and *hasAskedToWait* state variables to partition the above test templates further. Since *isLockedBy* is of type *OneThread*, it can only be empty or contain 1 element. When we do this, the result is $2 * 3 * 3 = 18$ test templates. At that stage, we decide not to partition the templates any further and we stop the process with these 18 leaf templates.

Applying the same type-based strategy to the other operations results in a total of 60 test templates for all the operations (all the other operations have fewer leaf templates than *lockRequestedByThread* because constraints on the valid input space of these operations restrict the number of possible combinations).

Instance and oracle templates. Consider the following leaf test template for *lockRequestedByThread*:

$$TT_{lockRequestedByThread} \hat{=} \left[VIS_{lockRequestedByThread} \mid \#isLockedBy = 1 \wedge \#isBlocking = 1 \wedge \#hasAskedToWait = 1 \right]$$

An instance template can be defined for this test template by instantiating the state variables and input:

$$\begin{aligned} IT_{lockRequestedByThread} &\hat{=} \\ &\left[isLockedBy : OneThread; isBlocking, hasAskedToWait : \mathbb{P} Thread; \right. \\ &\quad t? : Thread \mid t? = t_1 \wedge \\ &\quad \left. isLockedBy = \{t_2\} \wedge isBlocking = \{t_3\} \wedge hasAskedToWait = \{t_4\} \right] \end{aligned}$$

where t_1, t_2, t_3 and t_4 are all distinct threads in *Thread*.

The oracle/output template for this instance template is:

$$\begin{aligned} OT_{lockRequestedByThread} &\hat{=} \\ &\left[isLockedBy' : OneThread; isBlocking', hasAskedToWait' : \mathbb{P} Thread \mid \right. \\ &\quad \left. isLockedBy' = \{t_2\} \wedge isBlocking' = \{t_1, t_3\} \wedge hasAskedToWait' = \{t_4\} \right] \end{aligned}$$

Instance and oracle templates can easily be generated for the other 59 test templates as well.

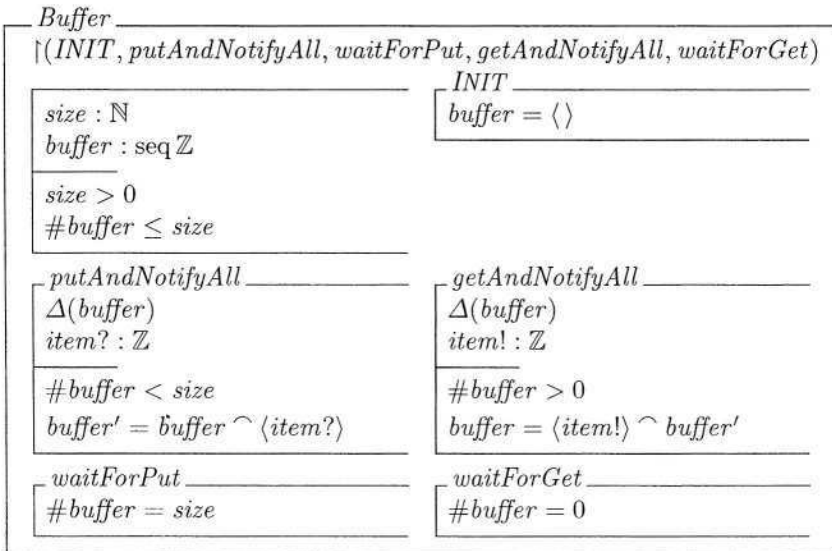
4 Concurrent Application Test Cases

4.1 Application Specification

Our approach to the specification of the application is to describe the different behaviours corresponding to the different thread paths through the synchronized object. This captures two important aspects of the application: (1) the effects on the application-specific variables, and (2) the conditions under which synchronisation occurs. The first aspect relates to the functional behaviour of the application that does not directly relate to its concurrent behaviour. That is, the effect on application-specific variables. The second aspect relates purely to the concurrent behaviour and captures the synchronisation policy of the application, i.e. the conditions under which the synchronisation primitives offered by the underlying concurrency mechanism should be invoked.

4.2 Example: Buffer

Our approach is illustrated by the following example specification of the Producer-Consumer problem. An Object-Z specification of the application-specific viewpoint of the finite buffer component presented in Figure 1 is now presented. The class provides the operations *putAndNotifyAll*, *waitForPut*, *getAndNotifyAll* and *waitForGet*. The internal mechanism of the buffer is modeled by a sequence. The *putAndNotifyAll* (and *getAndNotifyAll*) operation specifies the behaviour when a space for a resource (or a resource) is available in the buffer. These schemas also capture the conditions under which a *notifyAll* will be invoked. The *waitForPut* (and *waitForGet*) operation specifies the conditions in which a Producer (or Consumer) will wait until a put (or a get) is possible.



4.3 Buffer TTH

Test templates may be generated for the *Buffer* class by employing a type-based testing strategy. Variable *size* is examined by applying boundary analysis, e.g. the minimum size is 1 and we pick another “middle” size of 3. The specific values of the inputs and outputs may be left out of the test templates because the buffer is data-independent. Two examples follow from the 11 basic templates generated this way for the *Buffer* class.

$$TT_{putAndNotifyAll.1} \hat{=} [buffer : seq \mathbb{Z}; size : \mathbb{N} \mid \#buffer = 0 \wedge size = 1]$$

$$TT_{waitForPut.1} \hat{=} [buffer : seq \mathbb{Z}; size : \mathbb{N} \mid \#buffer = 1 \wedge size = 1]$$

$TT_{putAndNotifyAll.1}$ describes a case where a *put* operation should succeed and $TT_{waitForPut.1}$ describes a case where a *put* operation should wait.

Test instances and sequences. Test instances are generated from the templates by choosing appropriate values. Test oracles are created also. To enable test execution, test sequences must be created from the test instances. Test sequences have been generated automatically by bounded model checking using NuSMV [6]. Very briefly, we negate the test condition and get the model checker to produce the counter-example.

In practice, test sequences are comprised of calls to the Java component interface by different producer and consumer threads. Here, we use the interface offered by the Object-Z class. Sequences of executable method calls may be calculated from the operation sequences by taking into account the underlying JVM. (Some interactions may be infeasible in some versions of the JVM.) Example test sequences exercising test instances corresponding to the templates $TT_{putAndNotifyAll.1}$ and $TT_{waitForPut.1}$ follow.

$$(INIT(size = 1); putAndNotifyAll(item? = 1))$$

$$(INIT(size = 1); putAndNotifyAll(item? = 1); waitForPut)$$

Each test sequence begins with an *INIT* annotated with the size of the buffer (mimicking the Java constructor). The middle part of the sequence (which is empty for the first example) establishes the precondition of the desired test template. The last operation denotes a call to the operation in which the desired test template will be exercised.

While it is clear that checking the results of the execution of the test sequences against the oracles will verify the behaviour of the buffer with respect to the availability of the resource, the test sequences do not verify that the synchronisation mechanisms are called correctly. That is, we do not know whether *notifyAll* has been called rather than *notify*, or whether *wait* has been called when the *waitForPut*/*waitForGet* conditions occur. Verification of the suspension behaviour of the threads with respect to calls to *wait* and *notifyAll* is the topic of the next section.

5 A Concurrent Viewpoint of the Application

The complete specification of the concurrent behaviour of an application such as the buffer is formed by combining the application-specific behaviour as embodied in the *Buffer* class specified in the last section with the underlying concurrency model as embodied in the *Object* class specified in Section 2. It turns out that the complete specification of the behaviour involves not just application-specific detail but all of the detail of the synchronisation mechanism, and that test templates generated from the complete specification test the complete system, including the underlying synchronisation mechanism.

We will use the buffer example to first illustrate the problems inherent in testing the combination of application and synchronisation, and then see how to hide the internal mechanism of the *Object* class to produce test templates that focus solely on the application itself.

5.1 Buffer Object

The *Buffer* class and the *Object* class are combined to form the *BufferObject* class. It provides five (visible) operations.



The first operation, *lockRequestedByThread*, is inherited unchanged from the *Object* class. This operation corresponds to a thread requesting a lock for entry into a synchronised block. The operation *putAndNotifyAll* specifies the behaviour when a Producer thread successfully puts some item into the buffer. The specification of this operation makes direct use of the Object-Z inheritance mechanism. To be specific, as the definition of *putAndNotifyAll*, namely,

$$\textit{putAndNotifyAll} \hat{=} \textit{giveLockToThread} \ ; \ \textit{notifyAllThreads} \ ; \ \textit{lockReleasedByThread}$$

defined in the class *BufferObject* has the same name as an operation inherited from the class *Buffer*, it is conjoined with this inherited operation. The overall

result is that the specification of the operation *putAndNotifyAll* in the class *BufferObject* is equivalent to

$$\begin{aligned} \textit{putAndNotifyAll} \hat{=} & \textit{putAndNotifyAll}_{\textit{Buffer}} \\ & \wedge \\ & (\textit{giveLockToThread}_{\textit{Object}} \\ & \S \textit{notifyAllThreads}_{\textit{Object}} \\ & \S \textit{lockReleasedByThread}_{\textit{Object}}) \end{aligned}$$

where subscripts are used in this expression simply to indicate the class from which the specific operation is inherited. The overall effect of the *putAndNotifyAll* operation is that, a thread, having already requested entry into the synchronised block (by way of membership of the set *isBlocking*), receives the lock, notifies all waiting threads, releases the lock, and at the same time achieves the effect of the *putAndNotifyAll* operation specified in the *Buffer* class. This style of specification, used for each of the other operations in the *BufferObject* class, emphasises the concurrent aspects of the operation and helps delineate the synchronisation mechanism of *Object*.

The operation *waitForPut* specifies the behaviour when a put is not possible and the thread has to wait. Upon receiving the lock, the thread finds that the condition for putting an item, as described by the *waitForPut* operation inherited from *Buffer*, does not hold and hence the thread waits.

The operations *getAndNotifyAll* and *waitForGet* are similar.

5.2 Test Case Selection

Applying the Test Template Framework to this class is complicated by the use of sequential composition to specify the combined operations. However, a procedure [18] exists for creating the test templates and oracles of Object-Z operations formed by combining other operations with conjunction, sequential composition, and parallel composition, out of the test templates and oracles of the component operations. In [18], Periyasamy and Alagar consider compositions of operations from a single class and without inheritance. Single inheritance has been considered elsewhere [17]; however, our buffer-object example inherits from multiple classes.

Multiple inheritance requires that the test templates for the inherited operations are promoted to the complete inherited state. Strategies for further developing the inherited templates should be carefully chosen to fit the design of the application. This approach is demonstrated on the *lockRequestedByThread* and *putAndNotifyAll* operations below.

Example: *lockRequestedByThread*. This operation is inherited unchanged from *Object* and becomes an operation of the *BufferObject* class. However, the state of the *BufferObject* class consists of the state of the *Object* class merged with the state of the *Buffer* class. The approach presented in [18] for building test templates from sub-components is to start with the union of the test templates

of the sub-components, promote the test templates to the combined state-space and then apply further test strategies to expand the promoted result. For instance, to generate the test templates for the promoted *lockRequestedByThread* operation, one has to conjoin each test template in the test template hierarchy for *lockRequestedByThread* with a schema which describes what happens to the *Buffer* state during the *lockRequestedByThread* operation. However, the *lockRequestedByThread* operation does not change the state of the *Buffer* component. This results in a set of test templates similar to the following.

$$TT_{lockRequestedByThread.i} \hat{=} \exists Buffer.State \wedge Object.TT_{lockRequestedByThread.i}$$

Following this, one should apply test strategies to extend the test hierarchy further. A naive testing strategy for expanding this test hierarchy is to apply a type-based testing strategy to the promoted template.

If a 0-1-many testing strategy is applied to each promoted test template then the result will be $5 * 18 = 90$ test templates! However, these test templates are all re-testing the *lockRequestedByThread* operation in the presence of the application-specific inputs. This is testing two aspects.

1. It is re-testing *lockRequestedByThread*; exactly what we want to avoid, and
2. because the other operations, *putAndNotifyAll* etc. all rely on the given thread having already attempted entry to the synchronised block (as modelled by *lockRequestedByThread*) by way of the thread being in the set *isBlocking* (inherited from class *Object*), this is testing that a request for mutually exclusive access proceeds every other operation.

The first aspect should definitely be avoided, and more importantly, it is pointless to test the *lockRequestedByThread* for every application-specific input. However, by not testing *lockRequestedByThread*, there is a risk that exclusive access to the *Object* is not being verified.

In practice, it is impossible to test the correct use of *lockRequestedByThread* by black-box testing alone because the JVM manages the granting of locks by hidden internal operations. In light of this, the most sensible strategy for the black-box tester is to ignore the *lockRequestedByThread* operation¹.

Example: *putAndNotifyAll*. Applying the approach outlined above, the set of base test templates for *putAndNotifyAll* is the union of the promoted test templates of *putAndNotifyAll* from *Buffer* and the promoted test templates corresponding to the sequence of synchronisation operations from *Object*.

The test templates for the sequence may be generated using the procedure outlined in [18]. However, the promotion of the operations suffers from the same problems as described above. In addition, the sequence of operations from *giveLockToThread* to *lockReleasedByThread* of *Object* completely hides the granting of the lock (*isLockedBy* equals the empty set at the start and at the

¹ In practice, code inspection is a more effective way to check for the correct use of synchronised blocks and methods.

end). Furthermore, as is the case for *lockRequestedByThread*, the state of the set of blocked threads *isBlocking* is completely hidden by the JVM.

The problems illustrated above demonstrate the infeasability of testing the *Buffer* with the Test Template Hierarchy developed from the combination of *Buffer* and *Object* classes. This leads us to the conclusion that this deep combination of *Buffer* and *Object* is not appropriate for producing black-box tests and motivates the more abstract model of the *Object* now presented.

5.3 Restricted Object Viewpoint

We now consider a restricted viewpoint of *Object* that captures the use of synchronisation by the application but that does not retest the underlying mechanism. We observe that the application controls the membership of the set *hasAskedToWait* by use of *wait* and *notifyAll* but that the JVM controls the blocking of threads and the granting of locks by *giveLockToThread*. In addition, the application class does not specify a behaviour for the *lockRequestedByThread* operation because the effect of the associated entry into a synchronised block or method is completely hidden. It is the “use” of an object that forms the basis of our restricted viewpoint. Variable and operation hiding is used to restrict the test cases generated for class *Object*. The locked thread represented by variable *isLockedBy* and the related operation *lockRequestedByThread* are hidden as well as the blocking set represented by *isBlockedBy* and the related *giveLockToThread* and *lockReleasedByThread* operations. The class *UseObject* defines the resulting class and is expanded below.

$$UseObject \cong Object \setminus (isLockedBy, isBlocking, lockRequestedByThread, giveLockToThread, lockReleasedByThread)$$

<i>UseObject</i>	
$\{ (INIT, askThreadToWait, notifyThread, notifyAllThreads) \}$	
<i>hasAskedToWait</i> : $\mathbb{P} Thread$	<i>INIT</i> <i>hasAskedToWait</i> = $\emptyset$
<i>askThreadToWait</i> $\Delta(hasAskedToWait)$ <i>t!</i> : <i>Thread</i>	<i>notifyAllThreads</i> $\Delta(hasAskedToWait)$ <i>st!</i> : $\mathbb{P} Thread$
<i>hasAskedToWait'</i> = <i>hasAskedToWait</i> $\cup \{t!\}$	<i>st!</i> = <i>hasAskedToWait</i> <i>hasAskedToWait'</i> = $\emptyset$
<i>notifyThread</i>	
$\Delta(hasAskedToWait)$ <i>t!</i> : <i>OneThread</i>	
<i>t!</i> $\subseteq hasAskedToWait$ $\#hasAskedToWait > 0 \Rightarrow \#t! = 1$ <i>hasAskedToWait'</i> = <i>hasAskedToWait</i> $\setminus t!$	

UseObject Test Cases. The test templates resulting from applying the TTF to the *UseObject* class need only consider 0, 1, and many waiting threads. This gives 3 test templates for testing the correct application of each of the synchronisation mechanisms.

5.4 Application with UseObject

The application can be re-specified by the following.

<i>BufferUseObject</i>
$\{ (putAndNotifyAll, waitForPut, getAndNotifyAll, waitForGet) \}$
<i>Buffer</i>
<i>UseObject</i>
$putAndNotifyAll \hat{=} notifyAllThreads$
$waitForPut \hat{=} askThreadToWait$
$getAndNotifyAll \hat{=} notifyAllThreads$
$waitForGet \hat{=} askThreadToWait$

By including the definition $putAndNotifyAll \hat{=} notifyAllThreads$ explicitly in the class *BufferUseObject*, we are associating the name *putAndNotifyAll* with the operation *notifyAllThreads* inherited from *UseObject*. This ensures that this operation is conjoined with the operation *putAndNotifyAll* inherited from *Buffer*. The other three visible operations in the class *BufferUseObject* are defined similarly.

BufferUseObject Test Cases. The base test templates for the combination are the union of the test templates for *Buffer* and the test templates for *UseObject*. As illustrated below, a common strategy for developing the test hierarchy further is to consider the different types of threads.

6 Strategy

This section summarises our derived strategy for testing concurrent components.

1. Specify the application-specific viewpoint. Define operations that cover all synchronisation paths through the monitor. That is, paths that start when a lock is granted (resulting either from the initial entry or from being notified after waiting) and end when the lock is released (either from exiting the synchronised block or from waiting). Operations should differentiate between paths that use *notify*, *notifyAll*, or use no notification.
2. Partition the application-specific viewpoint operations into a base test template hierarchy.

3. Combine the test template hierarchy of the application-specific model with that of the restricted concurrency model to introduce thread suspension behavior.
4. Use the number and type of waiting threads to further develop the resultant test template hierarchy.

In step 1 we have used the underlying thread mechanism provided by the *Object* class to decide the synchronisation points that should be covered by the operations. In other work [13] we have used Concurrency Flow Graphs to produce the test conditions. In future work we will look at how these two approaches can be combined.

Step 2 is a standard application of the Test Template Framework to the viewpoint specification produced in Step 1.

Step 3 requires some ingenuity on the part of the tester to decide strategies that take advantage of the component design to avoid re-testing the underlying Java mechanism and to focus on the synchronisation under control of the application.

Step 4 is possible because of the introduction of the threads themselves in step 3. Test partitioning based on the number and type of suspended threads is standard for this type of application because it shows whether particular classes of thread (such as Producers or Consumers) are starved because of inappropriate wait conditions or notification.

As a further demonstration of this strategy we next apply it to the Readers-Writers problem.

6.1 Case Study: Readers-Writers

The Readers-Writers problem involves a shared resource which is read by reader threads and written to by writer threads. To prevent non-interference, individual writers must be given exclusive access to the resource, locking out other writers and readers. However, reading does not result in interference and so multiple readers may access the resource concurrently. A monitor is used to control access to the resource. Our approach is built on that presented in standard concurrency textbooks [2,16].

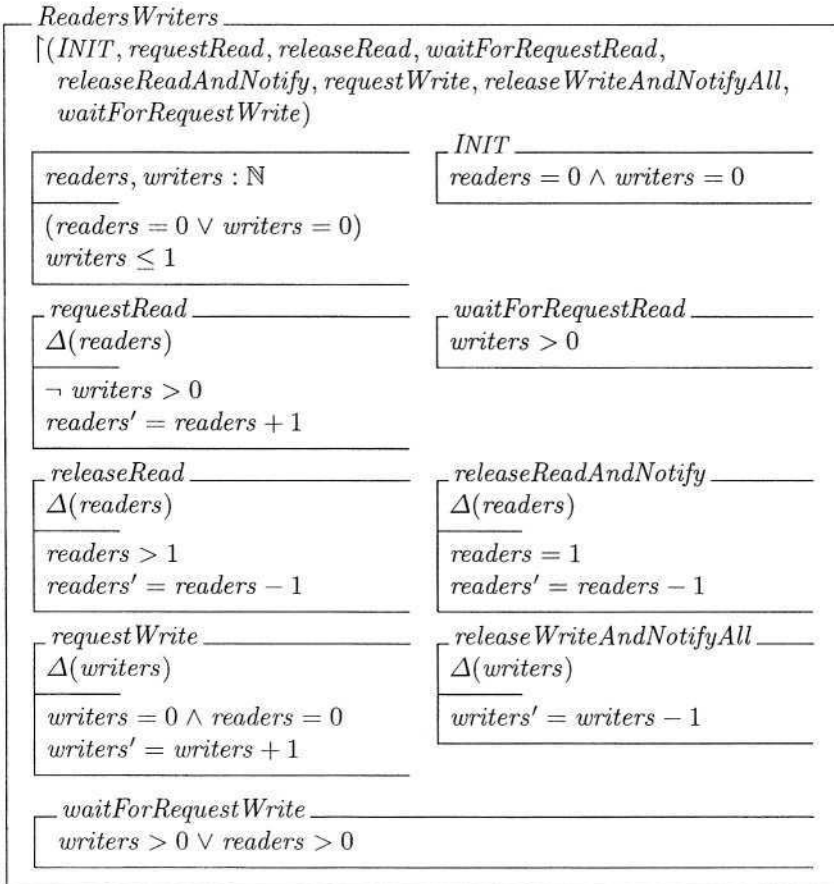
Step 1. We specify the application-specific behaviour of the Readers-Writers monitor by considering all paths through the synchronisation points. Object-Z class *ReadersWriters* (presented below) specifies operations corresponding to the different ways in which a thread may progress through the monitor.

The state of the monitor involves two counters corresponding to the number of *readers* and *writers* concurrently accessing the resource. The state invariant captures the desired *monitor invariant* the resource is accessed either by concurrent readers or writers but never by both, and the number of concurrent writers is never greater than one. Initially there are no readers or writers.

The operations capture the application-specific aspects only. The concurrent behaviour is added in Step 3.

A read request will succeed immediately if there are no writers. This is captured by operation *requestRead*. A read request will be delayed (the thread waits) if there is currently a writer accessing the resource (*waitForRequestRead*). Once finished reading, a reader releases the resource. There are two cases, if the thread is not the last reader, i.e., $readers > 1$ before the release, then the readers just stops reading as per *releaseRead*. However, if the reader is the last thread ($readers = 1$) then the reader notifies a waiting writer², as specified in *releaseReadAndNotify*.

A request for write access will succeed immediately if the number of readers and writers both equal 0 (*requestWrite*). A request for write access will wait otherwise (*waitForRequestWrite*). When releasing write access, a thread always notifies all other waiting threads (*releaseWriteAndNotifyAll*).



² It notifies any waiting thread but only writing threads will wait for a reader to release.

6.2 Step 2

Test templates are now generated for the application-specific viewpoint.

Many of the test templates correspond to the valid input space because the preconditions are so simple. In the cases of *requestRead* and *releaseRead* we apply a type-based strategy on the number of readers (0-1-many), and in the case of *waitForRequestWrite* we apply domain partitioning to the top-level disjunction and a type-based strategy; 11 base test templates are produced in this way.

6.3 Step 3

The application-specific viewpoint is now combined with the concurrency viewpoint capturing thread suspension behaviour. The class *ReadersWritersObject* describes the combination. It extends all *ReadersWriters* operations with the appropriate concurrent behaviour. As operations *requestRead*, *releaseRead*, and *requestWrite* all succeed immediately (without suspension) and do not notify any other threads, they do not need to be combined with any synchronisation operations. Operations *waitForRequestRead* and *waitForRequestWrite* are combined with *askThreadToWait* because they capture the waiting behaviour for a read or write request. Operation *releaseReadAndNotify* is combined with *notifyThread* because it captures the case when the last reading thread releases the resource and must notify a waiting writer (if one exists). Similarly, *releaseWriteAndNotifyAll* is combined with *notifyAllThreads* because all waiting threads must be notified when a writer releases the resource.

ReadersWritersObject _____
 [(*requestRead*, *waitForRequestRead*, *releaseRead*, *releaseReadAndNotify*,
requestWrite, *waitForRequestWrite*, *releaseWriteAndNotifyAll*)]

ReadersWriters

UseObject

waitForRequestRead $\hat{=}$ *askThreadToWait*
releaseReadAndNotify $\hat{=}$ *notifyThread*
waitForRequestWrite $\hat{=}$ *askThreadToWait*
releaseWriteAndNotifyAll $\hat{=}$ *notifyAllThreads*

Combined TTH. Test templates for the combined operations are constructed from the union of the test templates of the sub-operations. In the case of *requestRead* and other operations inherited directly from *ReaderWriters*, the test templates are just the ones inherited from the *ReadersWriters* test template hierarchy. In the case of the combined *waitForRequestRead*, *releaseReadAndNotify*, *waitForRequestWrite*, and *releaseWriteAndNotifyAll* operations, the combined test templates result from the union of test templates from the *ReadersWriters* class and those from the *UseObject* class.

6.4 Step 4

To further develop the test templates for the operations *waitForRequestRead*, *releaseReadAndNotify*, *waitForRequestWrite*, *releaseWriteAndNotifyAll*, we consider 0, 1, or many reader and writer threads waiting. This allows us to test, for instance, that all waiting threads are notified when *releaseWrite* is called.

7 Conclusion

While others have applied the Test Template Framework to interactive systems, in this paper we apply it to concurrent components. We have focused on the Java concurrency model but our approach could be generalised to other concurrency models such as protected Ada Objects.

Our approach has been to separate out the application and underlying concurrency mechanism into separate viewpoints and then to develop test hierarchies for them separately. We have then combined the test hierarchies for different viewpoints taking into account the designed isolation of the concurrency mechanism. This demonstrates a new approach to test template generation. In doing so we have had to deal with multiple inheritance, a previously untreated aspect of the application of the test template framework to object-oriented specifications.

Acknowledgments. This research is funded by an Australian Research Council Discovery grant, DP0343877: *Practical Tools and Techniques for the Testing of Concurrent Software Components*. This article has greatly benefited from proof-reading by, and discussion with, Doug Goldson and Brad Long.

References

1. Bernhard K. Aichernig. Test-case calculation through abstraction. In *Proceedings of Formal Methods Europe 2001, FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 571–589. Springer-Verlag, 2001.
2. G. Andrews. *Concurrent Programming: Principles and Practice*. Addison Wesley, 1991.
3. H. Bowman, M.W.A. Steen, E.A. Boiten, and J. Derrick. A formal framework for viewpoint consistency (full version). Computing Laboratory Technical Report 22-99, University of Kent at Canterbury, Canterbury, Kent, CT2 7NZ, December 1999.
4. Marius C. Bujorianu, Savi Maharaj, and Manuela Bujorianu. Towards a formalization of viewpoints testing. In Robert M. Hierons and Thierry Jéron, editors, *Formal Approaches To Testing of Software 2002 (FATES'02)*, Research Report, 35042 Rennes, France, August 2002. INRIA. A satellite workshop of CONCUR'02.
5. David Carrington, Ian MacColl, Jason McDonald, Leesa Murray, and Paul Strooper. From Object-Z Specifications to ClassBench Test Suites. *Software Testing, Verification and Reliability*, 10(2):111–137, 2000.

6. Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425, 2000.
7. R. Duke and G. Rose. *Formal Object-Oriented Specification Using Object-Z*. Cornerstones of Computing. Macmillan Press Limited, UK, 2000.
8. Roger Duke, Luke Wildman, and Brad Long. Modelling Java Concurrency with Object-Z. In A. Cerone and P. Lindsay, editors, *Software Engineering and Formal Methods (SEFM'03)*, pages 173–181. IEEE Computer Society Press, 2003.
9. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. Also online at <http://java.sun.com/docs/books/jls/index.html> as at Sep 2002.
10. Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, 1995.
11. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 2nd edition, 1999.
12. Brad Long, Dan Hoffman, and Paul Strooper. Tool support for testing concurrent Java components. *IEEE Transactions of Software Engineering*, 29(6):555–566, June 2003.
13. Brad Long and Paul Strooper. A Classification of Concurrency Failures in Java Components. In *Proceedings of the 1st International Workshop on Parallel and Distributed Systems: Testing and Debugging*, April 2003.
14. I. MacColl and D. Carrington. Testing matis: A case study on specification-based testing of interactive systems. In *Formal Aspects of HCI (FAHCI98)*, pages 57–69, 1998.
15. Ian Dugald MacColl. *Specification-Based Testing of Interactive systems*. PhD thesis, Information Technology and Electrical Engineering, The University of Queensland, Feb 2003.
16. J. Magee and J. Kramer. *Concurrency: State Models and Java Programs*. John Wiley & Sons, 1999.
17. L. Murray, D. Carrington, I. MacColl, and P. Strooper. Extending test templates with inheritance. In Paul A. Bailes, editor, *Proceedings of the Australian Software Engineering Conference ASWEC'97*, pages 80–87. IEEE Computer Society, September 1997.
18. K. Periyasamy and V.S. Alagar. A rigorous method for test templates generation from object-oriented specifications. *Software Testing, Verification and Reliability*, 11:3–37, 2001.
19. Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on software Engineering*, 22(11):777–793, November 1996.

A Method for Compiling and Executing Expressive Assertions

F.J. Galán Morillo and J.M. Cañete Valdeón

Dept. of Languages and Computer Systems. Faculty of Computer Science of Seville
Av. Reina Mercedes s/n 41012 Sevilla, Spain.
phone: (34) 95 455 27 73, fax: (34) 95 455 71 39
{galanm,canete}@lsi.us.es

Abstract. Programming with assertions constitutes an effective tool to detect and correct programming errors. The ability of executing formal specifications is essential in order to test automatically a program with respect to its assertions. However, formal specifications may describe recursive models which are difficult to identify so current assertion checkers limit, in a considerable way, the expressivity of the assertion language. In this paper, we are interested in showing how transformational synthesis can help to execute “expressive” assertions of the form $\forall x(r(x) \Leftrightarrow QyR(x,y))$ where x is a set of variables to be instantiated at execution time, Q is an existential or universal quantifier and R a quantifier free formula in the language of a particular first-order theory $\mathcal{A}$ we call assertion context. The class of assertion contexts is interesting because it presents a balance between expressiveness for writing assertions and existence of effective methods for executing them by means of synthesized (definite) logic programs.

Keywords: Assertion, correctness, formal specification, logic program, program synthesis, programming with assertions, meaning-preserving transformation, testing.

1 Introduction

Experience has shown that writing assertions while programming is an effective way to detect and correct programming errors. As an added benefit, assertions serve to document programs, enhancing maintainability. Programming languages such as Eiffel [18], SPARK [2] and recent extensions to the Java programming language, such as iContract [13], JML [16] and Jass [5], allow to write assertions into the program code in the form of pre-post conditions and invariants. Software components called assertion checkers are then used to decide if program assertions hold at execution time. However, and due to mechanization problems, current checkers do not accept the occurrence of unbounded quantification in assertions. This fact limits the expressivity of the assertion language and so the effectiveness of testing activities by means of runtime assertion checkers.

In order to motivate the problem, we show in Ex. 1 a (schematic) program which includes an unbounded quantified sub-formula in its post-condition.

Example 1. A program which returns a number different to zero if and only if the parameter l is a subset of the parameter s . Program types Nat and Set are used to represent natural numbers and sets of natural numbers respectively.

subset($l: Set, s: Set$) **return** $z: Nat$

pre: *true*

... *program code* ...

post: $\neg idnat(z, 0) \Leftrightarrow \forall e (member(e, l) \Rightarrow member(e, s))$

where *idnat* is the identity relation for natural numbers:

$idnat : Nat \times Nat$

$idnat(0, 0) \Leftrightarrow true$

$\forall x (idnat(s(x), 0) \Leftrightarrow false)$

$\forall y (idnat(0, s(y)) \Leftrightarrow false)$

$\forall x, y (idnat(s(x), s(y)) \Leftrightarrow idnat(x, y))$

where *member* is a relation to decide if a natural number is included or not in a set:

$member : Nat \times Set$

$\forall e (member(e, []) \Leftrightarrow false)$

$\forall e (member(e, [x|y]) \Leftrightarrow (idnat(x, e) \vee member(e, y)))$

At execution time, *subset*'s program code will supply values (ground terms) to assertion variables l , s and z closing their interpretations. Thus, the correctness of a program behavior such as *subset*($[s(0)], [0, s(0)]$) = $s(0)$ will depend on the evaluation of $\neg idnat(s(0), 0) \Leftrightarrow \forall e (member(e, [s(0)]) \Rightarrow member(e, [0, s(0)]))$ (i.e. *subset*'s post-condition after substituting l , s and z by values $[s(0)]$, $[s(0), 0]$ and $s(0)$ respectively). Due to the form of *idnat*'s axioms, it is not difficult to find a program able to evaluate ground atoms such as $idnat(s(0), 0)$, in fact, the if-part of *idnat*'s axioms can be considered one of such programs (Ex. 2). However, the occurrence of unbounded quantification in sub-formulas such as $\forall e (member(e, [s(0)]) \Rightarrow member(e, [0, s(0)]))$ complicates extraordinarily the search for such programs [9]. Due to this fact, current assertion checkers [18], [2], [5], [13], [16] does not consider the use of unbounded quantification in their assertion languages. Such a decision limits the expressivity of the assertion language and, therefore, the effectiveness of testing activities by means of runtime assertion checkers.

Example 2. A logic program which is able to evaluate ground atoms for *idnat*.

$idnat(0, 0) \Leftarrow$

$idnat(s(x), s(y)) \Leftarrow idnat(x, y)$

Our objective can be summarized in the following questions:

1. *Is it possible to extend current checkers to execute “expressive assertions”?*

Firstly, we need to formalize what we call “expressive assertions”. For us, an expressive assertion is a (new) relation r which represents a quantified

sub-formula $QyR(x, y)$ within a program assertion. Formally, r is defined by means of one axiom of the form $\forall x(r(x) \Leftrightarrow QyR(x, y))$ where x is a set of variables to be instantiated at execution time, Q is an existential or universal quantifier, and R a quantifier free formula in the language of a particular first order theory $\mathcal{A}$ called assertion context. For instance, $\forall l, s(r(l, s) \Leftrightarrow \forall e(member(e, l) \Rightarrow member(e, s)))$ is the definition of an expressive assertion r for the quantified sub-formula $\forall e(member(e, l) \Rightarrow member(e, s))$ in subset's post-condition (Ex. 1). Therefore, to answer "yes" to the question is equivalent to say that assertion checkers must be able to evaluate any ground atom $r(t)$ in $\mathcal{A}$ where t is the set of values supplied by the program at execution time.

2. How can we do it?

Logic program synthesis constitutes an important aid in order to overcome the problem. Our intention is:

- If $Q = \exists$, to synthesize a (definite) logic program $r_1^\exists$ of the form $r_1^\exists(x, y) \Leftarrow P^\exists(x, y)$ from a specification of the form $\forall x, y(r_1^\exists(x, y) \Leftrightarrow R(x, y))$.
- If $Q = \forall$, to synthesize a (definite) logic program $r_1^\forall$ of the form $r_1^\forall(x, y) \Leftarrow P^\forall(x, y)$ from a specification of the $\forall x, y(r_1^\forall(x, y) \Leftrightarrow \neg R(x, y))$.

In any case, synthesized programs r_1^Q must be *totally correct wrt goals of the form $\Leftarrow r_1^Q(t, y)$* . In order to synthesize logic programs, we have studied some of the most relevant synthesis paradigms (constructive, transformational and inductive) [7,8,9]. In particular, we are interested in transformational methods [4,14] however, some important problems are exposed in [6,8]: *"A transformation usually involves a sequence of unfolding steps, then some rewriting, and finally a folding step. The eureka about when and how to define a new predicate is difficult to find automatically. It is also hard when to stop unfolding. There is a need for loop-detection techniques to avoid infinite synthesis through symmetric transformations"*. In order to overcome these problems, we propose to develop program synthesis within assertion contexts [11]. Such a decision will allow us:

- To *structure* the search space for new predicates.
- To define a relation of similarity on formulas *for deciding when to define new predicates* and, from here, a particular folding rule *to define new predicates without human intervention*.
- To define an incremental compilation method where *no symmetric transformations are possible*.

3. How can assertion checkers evaluate ground atoms $r(t)$ from goals $\Leftarrow r_1^Q(t, y)$ in a refutation system such as Prolog?

- The execution of $\Leftarrow r_1^\exists(t, y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, \dots, \theta_j\}$ for y , thus:
If $\{\theta_1, \dots, \theta_j\} = \emptyset$, by total correctness of $r_1^\exists$, $\mathcal{A} \vdash \neg r(t)$ else $\mathcal{A} \vdash r(t)$.
- The execution of $\Leftarrow r_1^\forall(t, y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, \dots, \theta_j\}$ for y , thus:
If $\{\theta_1, \dots, \theta_j\} = \emptyset$, by total correctness of $r_1^\forall$, $\mathcal{A} \vdash r(t)$ else $\mathcal{A} \vdash \neg r(t)$.

Our work is explained in the following manner. In Sect(s). 2 and 3 we introduce a set of preliminary definitions and a brief background on transformational synthesis respectively. Section 4 formalizes assertion contexts as a class of particular first-order theories to write expressive assertions. Then, Sect. 5 defines a compilation method for expressive assertions. Section 6 explains how to execute expressive assertions from compilation results. Finally, we conclude in Sect. 7.

2 Preliminary Definitions

This section introduces a set of preliminary definitions in order to clarify the vocabulary we will use in the rest of the paper.

Definition 1 (Term, Formula). *A term of type τ is defined inductively as follows: (a) a variable of type τ is a term of type τ , (b) a constant of type τ is a term of type τ , and (c) if f is an n -ary function symbol of type $\tau_1, \dots, \tau_n \rightarrow \tau$ and t_i is a term of type τ_i ($i = 1..n$), then $f(t_1, \dots, t_n)$ is a term of type τ .*

A formula is defined inductively as follows: (a) if r is a relation symbol of type $\tau_1, \dots, \tau_n$ and t_i is a term of type τ_i ($i = 1..n$), then $r(t_1, \dots, t_n)$ is a typed atomic formula (or simply an atom), (b) if F and G are typed formulas, then so are $\neg F$, $F \wedge G$, $F \vee G$, $F \Rightarrow G$ and $F \Leftrightarrow G$ and (c) if F is a typed formula and x is a variable of type τ , then $\forall_\tau x F$ and $\exists_\tau x F$ are typed formulas (for legibility reasons we will omit subscripts in quantifiers). A typed literal is a typed atom or the negation of a typed atom.

A ground term (or value) is a term not containing variables. Similarly, a ground formula is a formula not containing variables.

A closed formula is a formula whose variables are quantified. A quantifier-free formula is a formula without quantifiers.

Definition 2 (Herbrand base). *The Herbrand universe of a first order language L is the set of all ground terms, which can be formed out of the constants and function symbols appearing in L . The Herbrand base for L is the set of all ground atoms which can be formed by using relation symbols from L with ground terms from the Herbrand universe of L .*

Definition 3 (Patterns). *A term pattern t_p is obtained from a term t by replacing each variable occurrence in t by the symbol $-$. An atom pattern $r(t)_p$ is obtained from an atom $r(t)$ by replacing every term occurrence in $r(t)$ by its respective term pattern. We say that l_p is a literal pattern for $r(t)$ if either $l_p = r(t)_p$ or $l_p = \neg r(t)_p$. A formula pattern F_p is obtained from a quantifier-free formula F by replacing every literal in F by its respective literal pattern.*

We say that $t_{1p} > t_{2p}$ if either $t_{1p} = -$ and $t_{2p} = f(t_{2,1p}, \dots, t_{2,np})$ or $t_{1p} = f(t_{1,1p}, \dots, t_{1,np})$ and $t_{2p} = f(t_{2,1p}, \dots, t_{2,np})$ and there exists a non-empty subset $S \subseteq \{1..n\}$ such that $t_{1,ip} > t_{2,ip}$ for every $i \in S$ and $t_{1,jp} = t_{2,jp}$ for every $j \in (\{1..n\} - S)$.

Let $r(t_{1,1}, \dots, t_{1,n})_p$ and $r(t_{2,1}, \dots, t_{2,n})_p$ be two atom patterns, we say that $r(t_{1,1}, \dots, t_{1,n})_p > r(t_{2,1}, \dots, t_{2,n})_p$ if there exists a non-empty set $S \subseteq \{1..n\}$ such that $t_{1,ip} > t_{2,ip}$ for every $i \in S$ and $t_{1,jp} = t_{2,jp}$ for every $j \in (\{1..n\} - S)$.

Definition 4 (Definite Logic Program, Definite Goal). A definite logic clause is a universally closed formula of the form $A \Leftarrow B_1, \dots, B_n$ where $A, B_1, \dots, B_n$ are atoms. A definite logic program is a finite set of definite program clauses (Ex. 2 shows a definite logic program). A definite goal is a clause of the form $\Leftarrow B_1, \dots, B_n$.

3 Background on Transformational Synthesis

In transformational synthesis, a sequence of meaning preserving transformation rules is applied to a specification until a program is obtained [8]. This kind of stepwise forward reasoning is feasible with axiomatic specifications of the form $\forall x(r(x) \Leftrightarrow R(x))$. There are atomic transformation rules such as unfolding (replacing an atom by its definition), folding (replacing a sub-formula by an atom) and rewrite and simplification rules. The objective of applying transformations is to filter out a new version of the specification where recursion may be introduced by a folding step. This usually involves a sequence of unfolding steps, then some rewriting, and finally a folding step. These atomic transformation rules constitute a correct and complete set for exploring the search space, however they lead to very tedious synthesis due to the no existence of a guiding plan, except for the objective of introducing recursion. The “eureka” about when and how to define a new predicate is difficult to find automatically. It is hard to decide when to stop unfolding and also there is a need for detecting symmetric transformations to avoid infinite synthesis.

4 Assertion Contexts

As we said in Sect. 1, an expressive assertion is written in the language of a particular first order theory called assertion context. In Ex. 3 we show assertion context $\mathcal{A}$ from which subset’s post-condition in Ex. 1 has been written. The Herbrand universe of $\mathcal{A}$ is formed out of the constants 0, [] and function symbols s and []. Every assertion in $\mathcal{A}$ (*idnat* and *member*) is formalized by means of a relation symbol, a signature and a finite set of first order axioms. The Herbrand base of $\mathcal{A}$ is the set of all ground atoms which can be formed by using relation symbols *idnat* and *member* with ground terms from the Herbrand universe of $\mathcal{A}$.

Example 3 (Assertion context from the specifier’s point of view).

Assertion Context $\mathcal{A}$

Nat generated by $0 : \rightarrow \text{Nat}, s : \text{Nat} \rightarrow \text{Nat}$

Set generated by $[] : \rightarrow \text{Set}, [] : \text{Nat} \times \text{Set} \rightarrow \text{Set}$

Assertion *idnat* : $\text{Nat} \times \text{Nat}$

$\text{idnat}(0, 0) \Leftrightarrow \text{true} \quad \forall x(\text{idnat}(s(x), 0) \Leftrightarrow \text{false})$

$\forall y(\text{idnat}(0, s(y)) \Leftrightarrow \text{false}) \quad \forall x, y(\text{idnat}(s(x), s(y)) \Leftrightarrow \text{idnat}(x, y))$

Assertion $member : Nat \times Set$

$$\forall e(member(e, [])) \Leftrightarrow false$$

$$\forall e, x, y(member(e, [x|y])) \Leftrightarrow (idnat(x, e) \vee member(e, y))$$

It is important to remark that assertion contexts must be processed by means of assertion checkers (i.e. software components) so reasonable restrictions have to be imposed on the form of axioms in order to make feasible their automatic processing. Example 4 shows $\mathcal{A}$ from the assertion checker's point of view where some redundant information (i.e. layers and patterns) are explicitly shown.

Example 4 (Assertion context from the assertion checker's point of view).

Assertion Context $\mathcal{A}$

Nat generated by $0 : \rightarrow Nat, s : Nat \rightarrow Nat$

Set generated by $[] : \rightarrow Set, [] : Nat \times Set \rightarrow Set$

Assertion $idnat : Nat \times Nat$

$$idnat(0, 0) \Leftrightarrow true \quad \forall x(idnat(s(x), 0) \Leftrightarrow false)$$

$$\forall y(idnat(0, s(y)) \Leftrightarrow false) \quad \forall x, y(idnat(s(x), s(y)) \Leftrightarrow idnat(x, y))$$

Assertion $member : Nat \times Set$

$$\forall e(member(e, [])) \Leftrightarrow false$$

$$\forall e, x, y(member(e, [x|y])) \Leftrightarrow (idnat(x, e) \vee member(e, y))$$

Layers.

layer 0: *idnat* **layer 1:** *member*

Patterns.

l-patterns: *idnat*(0, 0), *idnat*(0, *s*(-)), *idnat*(*s*(-), 0), *idnat*(*s*(-), *s*(-))
member(-, []), *member*(-, [|-])

i-patterns: *idnat*(-, 0), *idnat*(-, *s*(-)), *idnat*(0, -), *idnat*(*s*(-), -)

u-patterns: *idnat*(-, -), *member*(-, -)

For the purpose of writing consistent contexts, the following restrictions have been imposed on assertions:

1. Every axiom is of the form $\forall(a(x) \Leftrightarrow B(z))$ with $z \subseteq x$ where $a(x)$ is an atom called the left-hand side (lhs) of the axiom and $B(z)$ is a quantifier-free formula composed of literals and binary logical connectives called the right-hand side (rhs) of the axiom.
2. Every element in the Herbrand base of $\mathcal{A}$ unifies with the lhs of a unique axiom.
3. Every assertion is encapsulated in a layer. If a is located at layer i , a is a symbol of level i . Every positive atom occurring in $B(z)$ is defined on the symbol a or on a symbol of level $i - 1$ (if possible). Every negative atom occurring in $B(z)$ is defined on a symbol of level $i - 1$.
4. Every recursive specification is well-founded wrt a set of parameters.

Theorem 1 (Ground Decidability). *For every $a(t)$ in the Herbrand base of $\mathcal{A}$ either $\mathcal{A} \vdash a(t)$ or $\mathcal{A} \vdash \neg a(t)$. (A proof of this theorem can be found in [12]).*

From Theorem 1, we formalize the semantics of assertion contexts. Our proposal is borrowed from previous results in the field of deductive synthesis [3], [14], [15].

Definition 5 (Consistency). *A model for $\mathcal{A}$ is defined in the following terms:*

$$\mathcal{A} \models a(t) \text{ iff } \mathcal{A} \vdash a(t) \text{ and } \mathcal{A} \models \neg a(t) \text{ iff } \mathcal{A} \vdash \neg a(t)$$

for every $a(t)$ in the Herbrand base of $\mathcal{A}$.

For the purpose of structuring the search space for new predicates [8], assertion contexts are enriched by means of a set of atom patterns. We classify atom patterns into three categories: lower patterns (l-patterns), intermediate pattern (i-patterns) and upper patterns (u-patterns). A lower pattern is calculated from the atom on the lhs of an axiom and a upper pattern is calculated from an atom on the rhs of an axiom. The rest of atom patterns (i.e. intermediate patterns) are calculated from upper and lower patterns via $>$ (Sect. 2, Def.3): every intermediate pattern is lesser than any upper pattern and greater than any lower pattern.

5. Every atom occurring on the rhs of an axiom presents an intermediate atom pattern or a upper atom pattern.

For the purpose of legibility, we display atom patterns by means of directed graphs where atom patterns are nodes and directed links are instances of the relation $>$. For instance, Fig. 1 shows the set of atom patterns in $\mathcal{A}$ (Ex. 4).

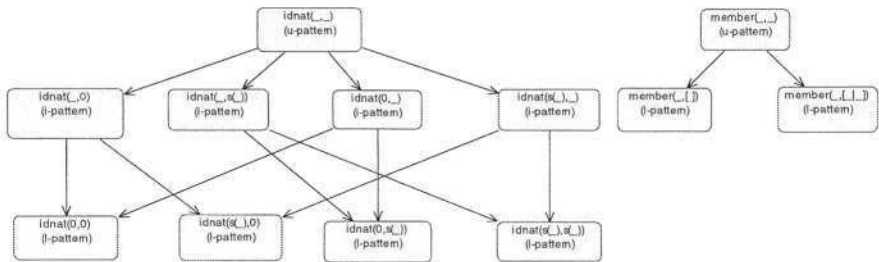


Fig. 1. Graph-based description of the set of atom patterns in $\mathcal{A}$.

Once we have formalized the notion of assertion context, we can formalize the notion of expressive assertion. Roughly speaking, an expressive assertion is a (new) relation r intended to represent a quantified sub-formula of the form $QyR(x, y)$ within a program assertion.

Definition 6 (Expressive Assertion). We say that r is an expressive assertion in $\mathcal{A}$ if and only if r is a new symbol not occurring in $\mathcal{A}$ which is defined by means of a unique axiom of the form $\forall x(r(x) \Leftrightarrow Q y R(x, y))$ where x is a set of variables to be instantiated at execution time, Q is a (existential or universal) quantifier and R is a quantifier-free formula in the language of $\mathcal{A}$ where every atom presents an intermediate atom pattern or an upper atom pattern in $\mathcal{A}$.

Example 5 (Expressive assertion for $\forall e(member(e, l) \Leftrightarrow member(e, s))$ in subset's post-condition (Ex. 1)).

$$\begin{aligned} \text{Assertion } r : Set \times Set \\ \forall l, s(r(l, s) \Leftrightarrow \forall e(member(e, l) \Rightarrow member(e, s))) \end{aligned}$$

5 Compilation Method

This section explains how transformational synthesis can help to compile an expressive assertion $\forall x(r(x) \Leftrightarrow Q y R(x, y))$. As we said in Sect. 1, our intentions are: If $Q = \exists$, to synthesize a *totally correct* (definite) logic program $r_1^\exists$ of the form $r_1^\exists(x, y) \Leftarrow P^\exists(x, y)$ from an auxiliary specification $\forall x, y(r_1^\exists(x, y) \Leftrightarrow R(x, y))$ and if $Q = \forall$, to synthesize a *totally correct* (definite) logic program $r_1^\forall$ of the form $r_1^\forall(x, y) \Leftarrow P^\forall(x, y)$ from an auxiliary specification $\forall x, y(r_1^\forall(x, y) \Leftrightarrow \neg R(x, y))$.

Example 6 (Auxiliary specification for r (Ex. 5)). In order to normalize the form of formulas (Def. 9), an equivalent formula $\forall(r_1^\forall(e, l, e, s) \Leftrightarrow \neg(member(e, l) \Rightarrow member(e, s)))$ is considered for $\forall(r_1^\forall(e, l, e, s) \Leftrightarrow (member(e, l) \wedge \neg member(e, s)))$.

$$\begin{aligned} \text{Assertion } r_1^\forall : Nat \times Set \times Nat \times Set \\ \forall e, l, s(r_1^\forall(e, l, e, s) \Leftrightarrow (member(e, l) \wedge \neg member(e, s))) \end{aligned}$$

A compilation is done by means of a finite sequence of meaning-preserving transformation steps. Each transformation step is composed of an expansion phase followed by a reduction phase. An expansion phase is intended to decompose a formula into a set of formulas and a reduction phase is intended to replace sub-formulas by new predicates. As we will show later, the set of new predicates (“recursive predicates”) is computable.

5.1 Expansion Phase

Expansion phase decomposes a formula F into a set of formulas by means of instantiations and unfolding steps. Our intention is to decompose F in a guided manner by using a particular rule called instantiation. To implement instantiations, we will use atom patterns in the following terms: if $r(y)$ is a selected atom to be instantiated in F and $r(y)_\mathcal{P}$ is a pattern which dominates a subgraph in the graph-based description of atom patterns for r then lower patterns in such a subgraph will induce a set of substitutions for variables in $r(y)$. Such sets of

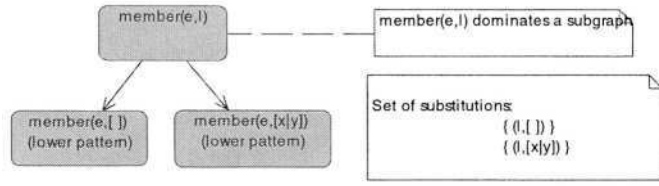


Fig. 2. Set of substitutions for $member(e, l)$.

substitutions will be the basis to construct instantiations. In Fig. 2 we show an example of a set of substitutions for an atom.

In Def(s). 7-9, we consider that F is a formula of the form $\forall(r_i(x) \Leftrightarrow R(x))$ with $R(x)$ as a quantifier-free formula written in (the language of an assertion context) $\mathcal{A}$.

Definition 7 (i-Instantiation). We say that $inst(F, i, a(y)) = \{\forall(r_i(x)\theta_1 \Leftrightarrow R(x)\theta_1), \dots, \forall(r_i(x)\theta_j \Leftrightarrow R(x)\theta_j)\}$ is the i -instantiation of an atom $a(y)$ in F if and only if

1. $a(y)$ is an atom in $R(x)$ of level i where $a(y)_P$ dominates a subgraph of patterns with lower patterns $\{a(y_1)_P, \dots, a(y_j)_P\}$.
2. $\{\theta_1, \dots, \theta_j\}$ is the set of substitutions such that $(a(y)\theta_k)_P = a(y_k)_P$ ($k = 1..j$).
3. Every atom in $R(x)\theta_k$ presents an atom pattern in $\mathcal{A}$.

Example 7 (Instantiation of $member(e, l)$ in the axiom of $r^\forall$ in Ex. 6).

$$\begin{array}{ll} \forall(r_i^\forall(e, [], e, s) \Leftrightarrow (member(e, []) \wedge \neg member(e, s))) & \theta_1 = \{([], [])\} \\ \forall(r_i^\forall(e, [x|y], e, s) \Leftrightarrow (member(e, [x|y]) \wedge \neg member(e, s))) & \theta_2 = \{([], [x|y])\} \end{array}$$

Definition 8 (Unfolding Step). Let $Ax = \forall(a(x) \Leftrightarrow B(z))$ be an axiom in an assertion context $\mathcal{A}$ and $a(y)$ an atom occurring in F with $a(x)\theta = a(y)$. We say that $unf(F, a(y), Ax)$ is the unfolding step of $a(y)$ in F wrt Ax if and only if $a(y)$ is replaced in F by $B(z)\theta$.

Definition 9 (Normalization Rules). To avoid negations in front of formulas, we normalize them by using the following set of rewrite rules where G and H are quantifier-free formulas.

(1) $\neg false \rightarrow true$,	(2) $\neg true \rightarrow false$
(3) $\neg\neg G \rightarrow G$,	(4) $\neg(G \Rightarrow H) \rightarrow (G \wedge \neg H)$
(5) $\neg(G \wedge H) \rightarrow (\neg G \vee \neg H)$	(6) $\neg(G \vee H) \rightarrow (\neg G \wedge \neg H)$
(7) $\neg(G \Leftrightarrow H) \rightarrow (\neg(G \Rightarrow H) \vee \neg(H \Rightarrow G))$	

Definition 10 (i-Expansion). We say that the set of formulas $exp(F, i)$ is the i -expansion of F if and only if every formula in $exp(F, i)$ is constructed by applying all i -instantiations (at least one) to F and then all unfolding steps (at least one) to each resulting formula. After unfolding steps, it can appear negative sub-formulas (i.e. presence of negation in front of unfolded sub-formulas). To avoid negations in front of such sub-formulas, we normalize them.

Example 8 (Expansion of the axiom for $r_1^\forall$ in Ex. 6).

$$\begin{array}{l}
 (1) \forall (r_1^\forall(e, [], e, [])) \Leftrightarrow (false \wedge true)) \\
 (2) \forall (r_1^\forall(e, [], e, [v|w])) \Leftrightarrow (false \wedge (\neg idnat(v, e) \wedge \neg member(e, w)))) \\
 (3) \forall (r_1^\forall(e, [x|y], e, [])) \Leftrightarrow ((idnat(x, e) \vee member(e, y)) \wedge true)) \\
 (4) \forall (r_1^\forall(e, [x|y], e, [v|w])) \Leftrightarrow ((idnat(x, e) \vee member(e, y)) \\
 \quad \wedge \\
 \quad (\neg idnat(v, e) \wedge \neg member(e, w))))
 \end{array}$$

5.2 Structuring the Search Space

Once an expressive assertion has been “decomposed” into a set of formulas (expansion), we are interested in *finding recursive compositions*. This can be done by identifying sub-formulas and replacing them by new predicates (reduction). Our intention is to anticipate and organize the search space for sub-formulas and new predicates in order to manage reductions automatically. Thus, after expanding an expressive assertion, we must be able for predicting the set of all possible resulting formulas.

Definition 11 (Search Space). Let $\forall x(r(x) \Leftrightarrow Q y R(x, y))$ be an expressive assertion in an assertion context $\mathcal{A}$ and $\forall x, y(r^Q(x, y) \Leftrightarrow R(x, y))$ the auxiliary specification from which a logic program has to be synthesized for r . We say that the set of formula patterns $\Omega(r)$ is the search space for r if and only if it includes the set of all formula pattern combinations which results of replacing all literals in R by atom patterns in $\mathcal{A}$ and by negative forms of atoms patterns in $\mathcal{A}$.

Every element $E_{\mathcal{P}}$ in $\Omega(r)$ encodes a sort of formulas. Such a codification depends on the sequence of relation symbols in $E_{\mathcal{P}}$. Our method considers that every formula pattern $E_{\mathcal{P}}$ is equivalent to a fresh atom pattern whose relation symbol, say $r_{E_{\mathcal{P}}}$, represents such a codification. In order to establish a precise codification, a bijection is proposed between term patterns in $E_{\mathcal{P}}$ and parameter positions in $r_{E_{\mathcal{P}}}$. We say that $\Omega_{ext}(r)$ is the *extended search space* for r if and only if $\Omega_{ext}(r)$ is formed from $\Omega(r)$ by including an element of the form $r_{E_{\mathcal{P}}} \Leftrightarrow E_{\mathcal{P}}$ for each element $E_{\mathcal{P}} \in \Omega(r)$. Thus, an extended search space represents a repository of new predicates (i.e. $r_{E_{\mathcal{P}}}$) and sub-formulas (i.e. $E_{\mathcal{P}}$) to be considered at reduction time.

Experimentally, it is important to note that no complete extended search spaces are needed when compiling expressive assertions. For instance, from a theoretical point of view, $|\Omega_{ext}(r_1^\forall)| = 196$ but only 9 of these patterns have been needed when compiling $r_1^\forall$ (Table 1). A practical result is proposed in [10] where we show that search spaces can be constructed on demand using tabulation techniques.

5.3 Similarity. A Criterion for Identifying New Predicates

In order to automate reductions, we propose a method to decide when a formula is similar to an element in a search space. We supply “operational” definitions to justify the mechanization of our proposal.

Table 1. $\Omega_{ext}(r_1^\forall)$ (partial).

(1) $r_1^\forall(-1, -2, -3, -4) \Leftrightarrow (member(-1, -2) \wedge \neg member(-3, -4))$
(2) $r_2^\forall(-1, -2) \Leftrightarrow (false \wedge \neg member(-1, -2))$
(3) $r_3^\forall(-1, -2) \Leftrightarrow (member(-1, -2) \wedge false)$
(4) $r_4^\forall(-1, -2) \Leftrightarrow (true \wedge \neg member(-1, -2))$
(5) $r_5^\forall(-1, -2) \Leftrightarrow (member(-1, -2) \wedge true)$
(6) $r_6^\forall \Leftrightarrow (false \wedge true)$
(7) $r_7^\forall \Leftrightarrow (false \wedge false)$
(8) $r_8^\forall \Leftrightarrow (true \wedge false)$
(9) $r_9^\forall \Leftrightarrow (true \wedge true)$

By $tree(R_P)$ we denote the *parse tree* of a formula pattern R_P where each leaf node contains a literal pattern and each internal node contains a binary logical connective. We say that a node in $tree(R_P)$ is *preterminal* if it has, at least, one leaf node.

We say that R_P is *similar wrt connectives* to E_P if and only if every binary logical connective in $tree(E_P)$ is located at the same place in $tree(R_P)$. In Fig. 3, we show that R_P is similar wrt connectives to E_P (but E_P is not similar wrt connectives to R_P). Similarity wrt connectives induces a mapping f from preterminal nodes in $tree(E_P)$ to subtrees in $tree(R_P)$ (for instance, f in Fig. 3).

Definition 12 (Similar Pattern). We say that a formula pattern R_P is *similar* to a formula pattern E_P if and only if

1. R_P is similar wrt connectives to E_P via mapping f ,
2. a) For each preterminal node $n \in tree(E_P)$ with two leaf nodes, e_{leftP} and e_{rightP} , there exist two leaf nodes, l_{leftP} on the left subtree of $f(n)$ and l_{rightP} on the right subtree of $f(n)$, where $e_{leftP} = l_{leftP}$ and $e_{rightP} = l_{rightP}$ and
- b) For each preterminal node $n \in tree(E_P)$ with one leaf node, e_{leftP}/e_{rightP} , there exists a leaf node l_{leftP}/l_{rightP} on the left/right subtree of $f(n)$, where $e_{leftP}/e_{rightP} = l_{leftP}/l_{rightP}$.

Figure 3 shows an example of similarity.

At this point, two interesting results are presented. The first one (Theorem 2) establishes that expansions preserve semantics and the second one (Theorem 3), which is needed to ensure termination, establishes that every formula resulting from an expansion can be reduced to a new predicate in the (extended) search space.

Theorem 2 (Expansion Preserves Correctness). Let $\{\forall(r_i(x_1) \Leftrightarrow R_1^{exp}(x_1)), \dots, \forall(r_i(x_j) \Leftrightarrow R_j^{exp}(x_j))\}$ be the set of resulting formulas in the i -expansion of a formula $\forall(r_i(x) \Leftrightarrow R(x))$. For every ground atom $r_i(x)\phi$ there exists a ground atom $r_i(x_k)\delta$ with $k \in \{1..j\}$ such that

$$\mathcal{A} \models R(x)\phi \Leftrightarrow R_k^{exp}(x_k)\delta$$

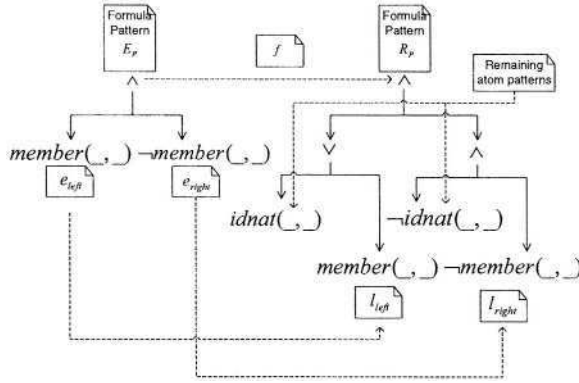


Fig. 3. R_P is similar to E_P .

(A proof of this theorem can be found in [12]).

Theorem 3 (Expansion is an Internal Operation in Ω_{ext}). Let $\{\forall(r_i(x_1) \Leftrightarrow R_1^{exp}(x_1)), \dots, \forall(r_i(x_j) \Leftrightarrow R_j^{exp}(x_j))\}$ be the i -expansion of a formula $\forall(r_i(x) \Leftrightarrow R(x))$. If $R(x)_P$ is similar to the rhs of some pattern in $\Omega_{ext}(r_i)$ then $R_k^{exp}(x_k)_P$ is also similar to the rhs of some pattern in $\Omega_{ext}(r_i)$ ($k = 1..j$). (A proof of this theorem can be found in [12]).

5.4 Reduction Phase

Reduction phase is intended to replace sub-formulas by atoms. To identify and replace sub-formulas by equivalent atoms are two key activities in a transformation step. Once a formula is similar to an element in a search space then it is rewritten (rewriting step), preserving its semantics, in order to facilitate an automatic replacement of sub-formulas by new predicates (folding step). In Def(s). 14, 15 and 16, we consider that F is a formula of the form $\forall(r_i(x) \Leftrightarrow R(x))$ with $R(x)$ a quantifier-free formula written in (the language of an assertion context) $\mathcal{A}$ and r a symbol not defined in $\mathcal{A}$.

Definition 13 (Simplification Rules). In order to simplify formulas in presence of propositions true and false, we consider the following set of rewrite rules where H is a formula.

(1) $\neg true \rightarrow false$	(2) $\neg false \rightarrow true$	(3) $true \vee H \rightarrow true$
(4) $false \vee H \rightarrow H$	(5) $true \wedge H \rightarrow H$	(6) $false \wedge H \rightarrow false$
(7) $false \Rightarrow H \rightarrow true$	(8) $true \Rightarrow H \rightarrow H$	(9) $false \Leftrightarrow H \rightarrow \neg H$
(10) $H \Rightarrow true \rightarrow true$	(11) $H \Rightarrow false \rightarrow \neg H$	(12) $true \Leftrightarrow H \rightarrow H$
(13) $\forall(true) \rightarrow true$	(14) $\forall(false) \rightarrow false$	

In the following definition, we use $R(a_1, \dots, a_p, \dots, a_n)$ to refer to $R(x)$ where $\{a_1, \dots, a_p, \dots, a_n\}$ is the set of all the atoms occurring in $R(x)$.

Definition 14 (Rewriting Step). Let P_p be a pattern in $\Omega_{ext}(r_i)$ such that $R(x)_p$ is similar to $E_p = rhs(P_p)$ with f as the induced mapping for deciding similarity wrt connectives and $\{a_1, a_2, \dots, a_p\}$ as the set of atoms in $R(a_1, \dots, a_p, \dots, a_n)$ which have not been used for deciding similarity (for instance, remaining atoms in Fig. 3). We say that $rew(F, P_p)$ is the rewriting step of F wrt P_p if and only if

(Step 1) We calculate the set of all the possible evaluations for $a_1, a_2, \dots, a_p$ in $R(x)$ in the following schematic manner:

$$rew(F, P_p) = \forall(r_i(x) \Leftrightarrow$$

$$\begin{array}{ll} (R(true, true, \dots, true, a_{p+1}, \dots, a_n) \wedge a_1 \wedge a_2 \wedge \dots \wedge a_p) & \vee \\ (R(false, true, \dots, true, a_{p+1}, \dots, a_n) \wedge \neg a_1 \wedge a_2 \wedge \dots \wedge a_p) & \vee \\ (R(true, false, \dots, true, a_{p+1}, \dots, a_n) \wedge a_1 \wedge \neg a_2 \wedge \dots \wedge a_p) & \vee \\ \dots & \vee \\ (R(false, false, \dots, false, a_{p+1}, \dots, a_n) \wedge \neg a_1 \wedge \neg a_2 \wedge \dots \wedge \neg a_p) & \end{array}$$

where each $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)$ denotes the replacement in $R(x)$ of the set of atoms $\{a_1, a_2, \dots, a_p\}$ by the combination $\{c_1, c_2, \dots, c_p\}$ of propositions true and false.

(Step 2) We simplify each $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)$ in the following form:

1. For each preterminal node $n \in tree(E_p)$ with two leaf nodes, we simplify (Def. 13) sub-formulas in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)$ which correspond to left and right subtrees of $f(n)$ in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)_p$.
2. For each preterminal node $n \in tree(E_p)$ with one left/right leaf node, we simplify (Def. 13) the sub-formula in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)$ which corresponds to the left/right subtree of $f(n)$ in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)_p$.

This selective simplification is intended to preserve similarity wrt connectives between $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)_p$ and E_p .

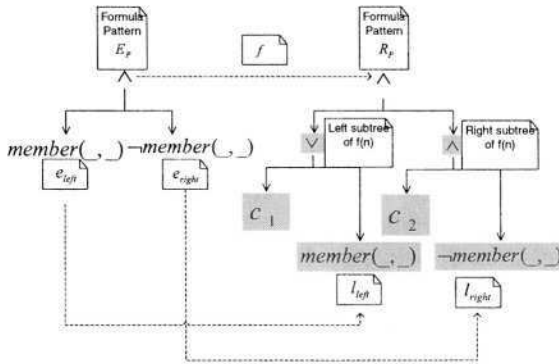


Fig. 4. Rewriting step. Sub-formulas to be simplified.

Example 9 (Rewriting Step). Let F be the formula (4) in Ex. 8, P_P the pattern (1) in $\Omega_{ext}(r_1^\forall)$ and $E_P = rhs(P_P)$. From Fig. 3 we can verify that $\{a_1 = idnat(x, e), a_2 = idnat(v, e)\}$ is the set of atoms which has not been used for deciding similarity (i.e. remaining atoms). After rewriting (step 1) we obtain the following formula:

$$\begin{aligned} rew(F, P_P) = \forall(r_1^\forall(e, [x|y], e, [v|w]) \Leftrightarrow \\ (true \vee member(e, y)) \wedge (false \wedge \neg member(e, w)) \wedge idnat(x, e) \wedge idnat(v, e) \vee \\ (false \vee member(e, y)) \wedge (false \wedge \neg member(e, w)) \wedge \neg idnat(x, e) \wedge idnat(v, e) \vee \\ ((true \vee member(e, y)) \wedge (true \wedge \neg member(e, w)) \wedge idnat(x, e) \wedge \neg idnat(v, e) \vee \\ (false \vee member(e, y)) \wedge (true \wedge \neg member(e, w)) \wedge \neg idnat(x, e) \wedge \neg idnat(v, e)) \end{aligned}$$

For preterminal node $\wedge$ in $tree(E_P)$, we simplify sub-formulas in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)$ which correspond to left and right subtrees of $f(\wedge)$ in $R(c_1, c_2, \dots, c_p, a_{p+1}, \dots, a_n)_P$. In Fig. 4 we have highlighted such subtrees. After rewriting (step 2), we obtain the formula:

$$\begin{aligned} rew(F, P_P) = \forall(r_1^\forall(e, [x|y], e, [v|w]) \Leftrightarrow \\ true \wedge false \quad \quad \quad \wedge idnat(x, e) \wedge idnat(v, e) \vee \\ member(e, y) \wedge false \quad \quad \quad \wedge \neg idnat(x, e) \wedge idnat(v, e) \vee \\ true \wedge \neg member(e, w) \quad \quad \quad \wedge idnat(x, e) \wedge \neg idnat(v, e) \vee \\ member(e, y) \wedge \neg member(e, w) \wedge \neg idnat(x, e) \wedge \neg idnat(v, e) \end{aligned}$$

In order to apply automatic folding to formulas, we need to instantiate patterns in extended search spaces. We say that a quantifier-free formula $pi(P_P, B)$ is the *pattern instantiation* of $P_P \in \Omega_{ext}(r_i)$ wrt B if and only if $B_P = rhs(P_P)$ and $pi(P_P, B)$ is obtained from P_P by replacing every term pattern in P_P by its respective term in B .

Example 10 (Pattern instantiation in $\Omega_{ext}(r_1^\forall)$).

$$\begin{aligned} P_P &= r_1^\forall(-_1, -_2, -_3, -_4) \Leftrightarrow (member(-_1, -_2) \wedge \neg member(-_3, -_4)) \\ B &= member(e, y) \wedge \neg member(e, w) \\ pi(P_P, B) &= r_1^\forall(e, y, e, w) \Leftrightarrow (member(e, y) \wedge \neg member(e, w)) \end{aligned}$$

Definition 15 (Folding Step). Let F be a formula in $\mathcal{A}$ of the form $\forall(r_i(x) \Leftrightarrow R(x))$ and B a sub-formula in $R(x)$ with $B_P = rhs(P_P)$ and $P_P \in \Omega_{ext}(r_i)$. We say that $fold(F, P_P)$ is the *folding step* of F wrt P_P if and only if it is obtained by replacing B by $lhs(pi(P_P, B))$ in $R(x)$.

Although search spaces are finite, to identify sub-formulas to be folded constitutes a highly non-deterministic task. In order to guide an automatic identification of such sub-formulas we introduce the notion of encapsulation and then explain how rewriting and folding rules contribute to automate reductions.

We say that a formula/formula pattern R/R_P is *completely encapsulated* in $layer_i$ of $\mathcal{A}$ if and only if every atom/atom pattern occurring in R/R_P is defined on a relation symbol of level i . We say that a formula/formula pattern R/R_P is *partially encapsulated* in $layer_i$ if and only if some atom/atom patterns occurring in R/R_P is defined on a relation symbol of level i and the remaining atom/atom patterns are defined on relation symbols of lower level.

Definition 16 (*i*-Reduction). Let $F_k \in \exp(F, i)$ be a formula of level i . The *i*-reduction of F_k wrt $\Omega_{ext}(r_i)$, $red(F_k, i, \Omega_{ext}(r_i))$, is implemented in the following steps:

1. (Searching). To search for patterns $P_P \in \Omega_{ext}(r_i)$ with $rhs(P_P)$ as a completely encapsulated pattern of level i . Literal patterns in $rhs(F_{kP})$ can be used to accelerate this search. If this search fails then to continue by searching for partially encapsulated patterns of level i . If this search fails then to continue in a similar way by searching for patterns of level $i - 1$ and so on.
2. (Rewriting, Step 1). Let $rhs(F_{kP})$ be similar to $rhs(P_P)$. We fix in $rhs(F_{kP})$ those atom patterns which are responsible of similarity and then a set A of remaining atoms in $rhs(F_k)$ is then selected to be evaluated.
3. (Rewriting, Step 2). After evaluating wrt A , we simplify by preserving the structure of logical connectives in P_P .
4. (Folding). To identify sub-formula B to be folded (i.e. $B_P = rhs(P_P)$), to construct a new predicate (i.e. $lhs(pi(P_P, B))$) and then to replace B in $rew(F_k, rhs(P_P))$ by the new predicate.

*Example 11 (*i*-Reduction).* Let F_k be the formula (4) in Ex. 8.

(Searching) We search for patterns $P_P \in \Omega_{ext}(r_1^\forall)$ such that $rhs(P_P)$ is a completely encapsulated pattern of level 1:

$$P_P = r_1^\forall(-_{-1,-2,-3,-4}) \Leftrightarrow (member(-_{-1,-2}) \wedge \neg member(-_{-3,-4}))$$

(Rewriting, Step 1) If $rhs(F_{kP})$ is similar to (the rhs of) several patterns then a non-deterministic choice must be done. In our example, the choice is deterministic (i.e. P_P is the unique candidate). We fix in $rhs(F_{kP})$ those atom patterns which are responsible of similarity.

$$rhs(F_{kP}) = (idnat(-, -) \vee \underline{member(-, -)}) \wedge (\neg idnat(-, -) \wedge \neg \underline{member(-, -)})$$

The set of remaining atoms $A = \{a_1 = idnat(x, e), a_2 = idnat(v, e)\}$ is then selected to be evaluated.

(Rewriting, Step 2) After evaluating and simplifying:

$rew(F_k, P_P) = \forall(r_1^\forall(e, [x y], e, [v w])) \Leftrightarrow$	
$true \wedge false$	$\wedge idnat(x, e) \wedge idnat(v, e) \vee$
$member(e, y) \wedge false$	$\wedge \neg idnat(x, e) \wedge idnat(v, e)) \vee$
$true \wedge \neg member(e, w)$	$\wedge idnat(x, e) \wedge \neg idnat(v, e) \vee$
$member(e, y) \wedge \neg member(e, w)$	$\wedge \neg idnat(x, e) \wedge \neg idnat(v, e)$

(Folding) At this point, it is easy to identify B as a sub-formula in $rew(F_k, P_P)$ whose pattern is equal to the $rhs(P_P)$.

$rew(F_k, P_P) = \forall(r_1^\forall(e, [x y], e, [v w])) \Leftrightarrow$	
$true \wedge false$	$\wedge idnat(x, e) \wedge idnat(v, e) \vee$
$member(e, y) \wedge false$	$\wedge \neg idnat(x, e) \wedge idnat(v, e)) \vee$
$true \wedge \neg member(e, w)$	$\wedge idnat(x, e) \wedge \neg idnat(v, e) \vee$
$member(e, y) \wedge \neg member(e, w)$	$\wedge \neg idnat(x, e) \wedge \neg idnat(v, e)$
$\underbrace{\hspace{10em}}_B$	

A new predicate is obtained by pattern instantiation (Ex. 10):

$$\boxed{lhs(pi(P_P, B)) = r_1^\forall(e, y, e, w)}$$

Finally, the replacement of B by the new predicate produces the formula:

$$\boxed{\begin{array}{l} \forall(r_1^\forall(e, [x|y], e, [v|w]) \Leftrightarrow \\ \quad true \wedge false \quad \wedge idnat(x, e) \quad \wedge idnat(v, e) \quad \vee \\ \quad member(e, y) \wedge false \quad \wedge \neg idnat(x, e) \wedge idnat(v, e)) \quad \vee \\ \quad true \wedge \neg member(e, w) \wedge idnat(x, e) \quad \wedge \neg idnat(v, e) \quad \vee \\ \quad \underbrace{r_1^\forall(e, y, e, w)}_{lhs(pi(P_P, B))} \quad \wedge \neg idnat(x, e) \wedge \neg idnat(v, e) \end{array}}$$

We say that an i -reduction $red(F_k, i, \Omega_{ext}(r_i))$ is *complete* when all possible folding steps have been applied to $rew(F_k, P_P)$. We say that a reduction phase has been *completed* for F if and only if a complete i -reduction has been applied to every formula in $exp(F, i)$.

Theorem 4 (Reduction Preserves Correctness). *Let $\forall(r_i(x) \Leftrightarrow R^{red}(x))$ be the i -reduction of an expanded formula $\forall(r_i(x) \Leftrightarrow R^{exp}(x))$ wrt $\Omega_{ext}(r_i)$. For every ground atom $r_i(x)\phi$,*

$$\mathcal{A} \models R^{red}(x)\phi \Leftrightarrow R^{exp}(x)\phi$$

(A proof of this theorem can be found in [12]).

5.5 Compilation as an Incremental and Terminating Process

The compilation of an axiom is completed by a finite sequence of meaning-preserving transformation steps. Each transformation step is composed of an expansion phase followed by a (complete) reduction phase. Table 2 shows $r_1^\forall$ (Ex. 6) after a transformation step.

Theorem 5 (Forms of Compiled Axioms). *After a transformation step, every resulting formula presents one of the following forms. (A proof of this theorem can be found in [12]):*

1. $\forall(r_i(x) \Leftrightarrow r_j(x))$ where $r_j(x)_P$ is equal to the lhs of some element in $\Omega_{ext}(r_i)$.
2. $\forall(r_i(x) \Leftrightarrow \bigvee r_j(x) \wedge G_j(x))$ where $r_j(x)_P$ is equal to the lhs of some element in $\Omega_{ext}(r_i)$ and $G_j(x)$ a conjunctive formula of literals defined on atoms whose patterns are included in $\mathcal{A}$.

Each transformation step represents an *increment* in the overall compilation process. Due to Theorem 5, each successive increment compiles either an axiom for r_j (e.g. $\forall(r_3^\forall(e, y) \Leftrightarrow (member(e, y) \wedge false))$) or an axiom for a new assertion from a literal in $G_j(x)$ (e.g. $\forall(r_{10}^\forall(x, e) \Leftrightarrow \neg idnat(x, e))$).

Theorem 6 (Termination). *The compilation of an expressive assertion is completed in a finite amount of increments. (A proof of this theorem can be found in [12]).*

For instance, the compilation of $r_1^\forall$ (Ex. 6) has been completed by means of 11 increments $(r_1^\forall, \dots, r_{11}^\forall)$.

Table 2. Compilation results after one transformation step for $r_1^\forall$ in Ex. 6.

(1) $\forall (r_1^\forall(e, [], e, []) \Leftrightarrow r_5^\forall)$			
(2) $\forall (r_1^\forall(e, [], e, [v w]) \Leftrightarrow (r_6^\forall$	$\wedge \text{idnat}(v, e)$	$\vee$	
	$r_2^\forall(e, w)$	$\wedge \neg \text{idnat}(v, e)))$	
(3) $\forall (r_1^\forall(e, [x y], e, []) \Leftrightarrow (r_9^\forall$	$\wedge \text{idnat}(x, e)$	$\vee$	
	$r_5^\forall(e, y)$	$\wedge \neg \text{idnat}(x, e)))$	
(4) $\forall (r_1^\forall(e, [x y], e, [v w]) \Leftrightarrow (r_7^\forall$	$\wedge \text{idnat}(x, e)$	$\wedge \text{idnat}(v, e)$	$\vee$
	$r_3^\forall(e, y)$	$\wedge \neg \text{idnat}(x, e)$	$\wedge \text{idnat}(v, e)$
	$r_4^\forall(e, w)$	$\wedge \text{idnat}(x, e)$	$\wedge \neg \text{idnat}(v, e)$
	$r_1^\forall(e, y, e, w) \wedge \neg \text{idnat}(x, e)$	$\wedge \neg \text{idnat}(v, e)))$	$\vee$

6 Executing Expressive Assertions from Synthesized Logic Programs

Once a compilation process has been completed, a finite set of new assertions have been produced (Theorem 6). The form of their axioms (i.e. universal closure, mutually-exclusive disjunctions of conjunctions, absence of negated atoms and stratification [17]) allow to define a simple translation method from compiled assertions to definite logic programs.

Definition 17 (Translation Method). *For every resulting axiom Ax from a compilation:*

1. *If Ax is of the form $\forall(r \Leftrightarrow P)$ where P is a propositional formula formed out from constants true and false, two situations are possible:*
 - a) *If the evaluation of P is equal to false then Ax is translated to an empty clause.*
 - b) *If the evaluation of P is equal to true then Ax is translated to a clause of the form $r \Leftarrow$*
2. *If Ax is of the form $\forall(r(x) \Leftrightarrow \bigvee_k (r_1(x) \wedge \dots \wedge r_n(x)))$ then it is translated to a set of k clauses of the form $r(x) \Leftarrow r_1(x), \dots, r_n(x)$. Every clause, which includes an atom occurrence r with axiom of the form $\forall(r \Leftrightarrow P)$ and P equal to false, is deleted.*

Example 12 (Synthesized logic program for $r_1^\forall$ (Ex. 6)).

$$\begin{aligned}
 r_1^\forall(e, [], e, [v|w]) &\Leftarrow r_2^\forall(e, w), r_{10}^\forall(v, e) \\
 r_1^\forall(e, [x|y], e, []) &\Leftarrow r_{11}^\forall(x, e) \\
 r_1^\forall(e, [x|y], e, []) &\Leftarrow r_5^\forall(e, y), r_{10}^\forall(x, e) \\
 r_1^\forall(e, [x|y], e, [v|w]) &\Leftarrow r_3^\forall(e, y), r_{10}^\forall(x, e), r_{11}^\forall(v, e) \\
 r_1^\forall(e, [x|y], e, [v|w]) &\Leftarrow r_4^\forall(e, w), r_{11}^\forall(x, e), r_{10}^\forall(v, e) \\
 r_1^\forall(e, [x|y], e, [v|w]) &\Leftarrow r_1^\forall(e, y, e, w), r_{10}^\forall(x, e), r_{10}^\forall(v, e) \\
 r_2^\forall(e, [x|y]) &\Leftarrow r_2^\forall(e, y), r_{10}^\forall(x, e) \\
 r_3^\forall(e, [x|y]) &\Leftarrow r_3^\forall(e, y), r_{10}^\forall(x, e)
 \end{aligned}$$

$$r_4^\forall(e, []) \Leftarrow r_9^\forall$$

$$r_4^\forall(e, [v|w]) \Leftarrow r_4^\forall(e, w), r_{10}^\forall(v, e)$$

$$r_5^\forall(e, [v|w]) \Leftarrow r_{11}^\forall(v, e)$$

$$r_5^\forall(e, [v|w]) \Leftarrow r_5^\forall(e, w), r_{10}^\forall(v, e)$$

$$r_9^\forall \Leftarrow$$

$$r_{10}^\forall(0, s(y)) \Leftarrow$$

$$r_{10}^\forall(s(x), 0) \Leftarrow$$

$$r_{10}^\forall(s(x), s(y)) \Leftarrow r_{10}^\forall(x, y)$$

$$r_{11}^\forall(0, 0) \Leftarrow$$

$$r_{11}^\forall(s(x), s(y)) \Leftarrow r_{11}^\forall(x, y)$$

For the purpose of verifying that synthesized logic programs are totally correct wrt goals $\Leftarrow r_1^Q(t, y)$, a set of execution modes can be calculated for each synthesized predicate. An execution mode is formed by replacing each parameter in a signature by a mark $\downarrow$ to refer to ‘a ground term as input parameter’ or $\uparrow$ to refer to ‘an existentially quantified variable as output parameter’. For instance, a logic program such as the one shown in Ex. 2 is a totally correct program for *idnat* (Ex. 3) wrt goals $\Leftarrow idnat_1^\forall(t, y)$ (i.e. $idnat(\downarrow, \uparrow)$), $\Leftarrow idnat_1^\forall(x, t)$ (i.e. $idnat(\uparrow, \downarrow)$) and $\Leftarrow idnat_1^\forall(t_1, t_2)$ (i.e. $idnat(\downarrow, \downarrow)$) where t, t_1, t_2 are ground *Nat*-terms. Static analysis techniques can be used to calculate and/or verify sets of execution modes for a logic program [1], [6]. Table 3 shows the set of execution modes calculated for the synthesized program in Ex. 12. Thus, if a synthesized logic program $\forall x, y (r_1^Q(x, y) \Leftarrow P(x, y))$ for an expressive assertion $\forall x (r(x) \Leftrightarrow Q y R(x, y))$ presents $r_1^Q(\downarrow, \uparrow)$ as one of its execution modes then r_1^Q can be used to execute ground instances of r . For instance, such a condition holds for $r_1^\forall$ (Table 3), hence, the synthesized logic program in Ex. 12 can be used to execute ground instances of r in Ex. 5.

Table 3. Execution modes for the synthesized program in Ex. 12.

$r_{11}^\forall(\downarrow, \downarrow)$	$r_{11}^\forall(\uparrow, \downarrow)$	$r_{11}^\forall(\downarrow, \uparrow)$
$r_{10}^\forall(\downarrow, \downarrow)$	$r_{10}^\forall(\uparrow, \downarrow)$	$r_{10}^\forall(\downarrow, \uparrow)$
$r_5^\forall(\downarrow, \downarrow)$	$r_5^\forall(\uparrow, \downarrow)$	
$r_4^\forall(\downarrow, \downarrow)$	$r_4^\forall(\uparrow, \downarrow)$	
$r_3^\forall(\downarrow, \downarrow)$	$r_3^\forall(\uparrow, \downarrow)$	
$r_2^\forall(\downarrow, \downarrow)$	$r_2^\forall(\uparrow, \downarrow)$	
$r_1^\forall(\downarrow, \downarrow, \downarrow, \downarrow)$	$r_1^\forall(\uparrow, \downarrow, \uparrow, \downarrow)$	

How can assertion checkers evaluate ground atoms $r(t)$ from goals $\Leftarrow r_1^Q(t, y)$ in a refutation system such as Prolog?

The execution of $\Leftarrow r_1^Q(t, y)$ in a Prolog system will compute a set of substitutions $\{\theta_1, \dots, \theta_j\}$ for y , thus:

- For $Q = \exists$:
 1. If $\{\theta_1, \dots, \theta_j\} = \emptyset$ then $\mathcal{A} \vdash \neg \exists y(r_1^\exists(t, y))$ (by logical consequence), $\mathcal{A} \vdash \neg \exists y(R(t, y))$ (by total correctness of $r_1^\exists$), $\mathcal{A} \vdash \neg r(t)$ (by equivalence)
 2. If $\{\theta_1, \dots, \theta_j\} \neq \emptyset$ then, by a similar reasoning, $\mathcal{A} \vdash \exists y(r_1^\exists(t, y))$, $\mathcal{A} \vdash \exists y(R(t, y))$, $\mathcal{A} \vdash r(t)$.
- For $Q = \forall$:
 1. If $\{\theta_1, \dots, \theta_j\} = \emptyset$ then $\mathcal{A} \vdash \neg \exists y(r_1^\forall(t, y))$ (by logical consequence), $\mathcal{A} \vdash \neg \exists y(\neg R(t, y))$ (by total correctness of $r_1^\forall$), $\mathcal{A} \vdash \forall y(R(t, y))$ (by equivalence), $\mathcal{A} \vdash r(t)$ (by equivalence).
 2. If $\{\theta_1, \dots, \theta_j\} \neq \emptyset$ then, by a similar reasoning, $\mathcal{A} \vdash \exists y(r_1^\forall(t, y))$, $\mathcal{A} \vdash \exists y(\neg R(t, y))$, $\mathcal{A} \vdash \neg \forall y(R(t, y))$, $\mathcal{A} \vdash \neg r(t)$.

7 Conclusions and Future Work

In this paper, we have formalized a class of assertions we call expressive assertions in the sense that they describe recursive models which are not directly translatable into executable forms. Due to this fact, current assertion checkers are not able to execute expressive assertions. The existence of mature studies in the field of transformational synthesis constitutes an important aid to overcome the problem. Recurrent problems in transformational synthesis have been revisited, for instance, the “eureka problem” (i.e. non-automatic steps about when and how to define recursive predicates). In order to overcome the problem, we restrict our attention to a particular class of first-order theories we call assertion contexts. This sort of theories is interesting because it presents a balance between expressiveness for writing assertions and existence of effective methods for compiling and executing them via synthesized (definite) logic programs.

Finally, we consider that our work can also be used to construct assertion contexts in an incremental manner. In fact, assertions contexts can be extended with expressive assertions in a conservative way without losing execution capabilities. For instance, $\mathcal{A} \cup r \cup r_1^\forall$ is more expressive than $\mathcal{A}$ while preserving consistency and execution capabilities (i.e. $r_1^\forall$ can be used to execute ground atoms of r). This issue is essential from a practical view point in order to reach expressive assertion languages. We plan to study it as future work.

References

1. Arts, T., Zantema, H.: Termination of Logic Programs Using Semantic Unification. LOPSTR’95. Springer-Verlag, (1996) 219–233.
2. Barnes, J.: High Integrity Ada: The SPARK Approach. Addison-Wesley, (1997).
3. Bertoni, A., Mauri G., Miglioli, P.: On the Power of Model Theory in Specifying Abstract Data Types and in capturing their Recursiveness. *Fundamenta Informaticae*, **VI(2)** (1983) 27–170.
4. Burstall, R. M., Darlington, J.: A Transformational System for Developing Recursive Programs. *Journal of the ACM* **24(1)** (1977) 44–67.
5. Bartetzko, D., Fischer, C., Möller, M., Wehrheim, H.: Jass-Java with Assertions. 1st Workshop on Runtime Verification. Paris. France. ENTCS Elsevier, (1999).

6. Deville, Y.: Logic Programming. Systematic Program Development. Addison-Wesley. (1990)
7. Deville, Y., Lau, K. K.: Logic Program Synthesis. *J. Logic Programming* **19,20** (1994) 321–350.
8. Flener, P.: Logic Program Synthesis from Incomplete Information. Kluwer Academic Publishers, Massachusetts, (1995).
9. Flener, P.: Achievements and Prospects of Program Synthesis. *LNAI* 2407. Springer-Verlag, (2002) 310–346.
10. Galán, F. J., Cañete, J. M.: Improving Constructive Synthesizers by Tabulation Techniques and Domain Ordering. In David Warren (ed.), *Tabulation and Parsing Deduction*, (2000) 37–49.
11. Galán, F. J., Díaz, V. J., Cañete, J. M.: Towards a Rigorous and Effective Functional Contract for Components. *Informatica. An International Journal of Computing and Informatics*. 25(4), (2001) 527–533.
12. Galán, F. J., Cañete, J. M.: Compiling and Executing Assertions via Synthesized Logic Programs. Technical Report LSI-2004-01. Dept. of Language and Computer Systems. Faculty of Computer Science, Univ. of Seville. (2004)
13. Kramer, R.: iContract-The Java Design by Contract Tool. *TOOLS 26: Technology of Object-Oriented Languages and Systems*. IEEE Computer Society Press, (1998).
14. Lau, K., Ornaghi, M.: On Specification Frameworks and Deductive Synthesis of Logic Programs. *LOPSTR'94. LNCS 883* Springer-Verlag, (1994) 104–121.
15. Lau, K., Ornaghi, M.: Towards an Object-Oriented Methodology for Deductive Synthesis of Logic Programs. *LOPSTR'95. LNCS 1048* Springer-Verlag, (1995) 152–169.
16. Leavens, G., Baker, A., Ruby, C. Preliminary Design of JML. TR 98-06u, Dept. of Computer Science, Iowa State Univ., USA (2003).
17. Lloyd, J. W.: *Foundations of Logic Programming*, 2nd ed. Springer-Verlag, (1987).
18. Meyer, B.: *Eiffel: The Language*. Prentice-Hall, (1992).

Author Index

- Androutsopoulos, K. 187
Arias, José J. Pazos 382
- Ball, Thomas 1
Beckert, Bernhard 207
Blandford, Ann 461
Boulton, Richard J. 21
Boute, Raymond 441
Bujorianu, Marius C. 421
- Cañete Valdeón, J.M. 521
Cavalcanti, Ana 40
Chaki, Sagar 128
Chen, Yifeng 402
Ciobanu, Gabriel 307
Clark, D. 187
Clarke, Edmund M. 128
Cook, Byron 1
Curzon, Paul 461
- Díaz Redondo, Rebeca P. 382
Dong, Jin Song 168
Duke, Roger 501
Dunne, Steve 328
- Ellis, Bill J. 67
- Farias, Adalberto 108
Fernández Vilas, Ana 382
- Galán Morillo, F.J. 521
García Duque, Jorge 382
Geguang, Pu 363
Gottliebsen, Hanne 21
- Hardy, Ruth 21
Hung, Dang Van 363
- Ireland, Andrew 67
- Jézéquel, Jean-Marc 481
Jifeng, He 363
- Kelsey, Tom 21
Klaudel, Hanna 287
- Lano, K. 187
Lazić, R.S. 247
Levin, Vladimir 1
Liu, Zhiming 402
Lucanu, Dorel 307
- Martin, Ursula 21
Melham, Tom 36
Möller, Michael 267
Mota, Alexandre 108
- Olderog, Ernst-Rüdiger 267
Ouaknine, Joël 128
- Pickin, Simon 481
Plosila, Juha 227
- Qin, Shengchao 168
- Rajamani, Sriram K. 1
Rasch, Holger 267
Roscoe, A.W. 247
- Sampaio, Augusto 108
Schlager, Steffen 207
Schneider, Steve 87
Seculeanu, Tiberiu 227
Sharygina, Natasha 128
Sinha, Nishant 128
Solla, Alberto Gil 382
Strooper, Paul 501
Sun, Jun 168
- Thanh, Cécile Bui 287
Treharne, Helen 87
- Wang, Xu 247
Wehrheim, Heike 267
Wildman, Luke 501
Willemse, Tim A.C. 343
Winter, Kirsten 148
Woodcock, Jim 40
- Yi, Wang 363