Egon Börger
Angelo Gargantini
Elvinia Riccobene (Eds.)

LNCS 2589

Abstract State Machines 2003

Advances in Theory and Practice

10th International Workshop, ASM 2003
Taormina, Italy, March 2003
Proceedings

Springer

Lecture Notes in Computer Science 2589

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

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Egon Börger Angelo Gargantini
Elvinia Riccobene (Eds.)

Abstract State Machines 2003

Advances in Theory and Practice

10th International Workshop, ASM 2003
Taormina, Italy, March 3-7, 2003
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Egon Börger
Università di Pisa, Dipartimento di Informatica
Via Buonarotti 2, 56127 Pisa, Italy
E-mail: boerger@di.unipi.it

Angelo Gargantini
Università di Catania, CEA
Piazza Università 2, 95124 Catania, Italy
E-mail: gargantini@dmi.unict.it

Elvinia Riccobene
Università di Catania, Dipartimento di Matematica e Informatica
Città Universitaria, viale A. Doria 6, 95125 Catania, Italy
E-mail: riccobene@dmi.unict.it

Cataloging-in-Publication Data applied for

A catalog record for this book is available from the Library of Congress

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliographie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

CR Subject Classification (1998): F.3, D.2, F.4.1, D.3, H.2.3

ISSN 0302-9743
ISBN 3-540-00624-9 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2003
Printed in Germany

Typesetting: Camera-ready by author, data conversion by DA-TeX Gerd Blumenstein
Printed on acid-free paper SPIN: 10872467 06/3142 5 4 3 2 1 0

ASM 2003—A Double Anniversary in Sicily

ASM 2003 and this volume constitute a double landmark in the short history of the Abstract State Machine method for the design and the analysis of complex software/hardware systems. The volume contains the Proceedings of the 10th *International Workshop on Abstract State Machines* (ASM 2003), held in Taormina (Sicily) from March 3rd to March 7th, 2003. The anniversary edition of this workshop was the first one held in Italy, the country where Yuri Gurevich, in his influential 1986 and 1990 Spring lectures in Pisa, explained for the first time his ideas about generalizing Turing's thesis by a resource-bound-aware model of computation[1], and the country where the first of an ever growing series of European Ph.D. and habilitation theses was written (1989–1992)[2] which apply ASMs to real-life computing systems or develop their theoretical foundation further. It was also 10 years ago, in Sicily, that the fundamental Lipari Guide[3] saw the light, namely at the Lipari Summer School which in 1993 was devoted to *Specification and Validation Methods*.

The previous editions of the international workshop series on Abstract State Machines were held in the following European cities: Dagstuhl, Germany (2002); Las Palmas de Gran Canaria, Spain (2001); Monte Verità, Switzerland (2000); Toulouse, France (1999); Magdeburg, Germany (1998); Cannes, France (1998, 1997); Paderborn, Germany (1996); and Hamburg, Germany (1994).

ASM 2003 aimed at enforcing two valuable traditions of ASM workshops. In the first place ASM 2003 was devoted to both the theory and the multiple real-life applications of ASMs, with the goal to provide a forum for a survey and a critical evaluation of the current academic and industrial developments of the ASM method, aiming at a fruitful interaction between theory and practice. As organizers we were eager to maintain at ASM 2003 the vivid, frank but fair style of scientific disputation which has characterized past editions of the workshop.

The second concern of ASM 2003 was to offer a moment of reflection upon the place the ASM method occupies in the field of Computer Science, paying attention in particular to the relation of the method to similar or complementary system development and analysis approaches. This was also one of the reasons why in the past half of the ASM workshops were held as part of larger computer science conferences: ASM 2001 as part of Eurocast'01, ASM 1999 as part of FME'99, ASM 1998 in Magdeburg as part of GI-Jahrestagung, and ASM 1994 as part of the IFIP World Congress. The goal is to pave the way for an integration of

[1] For historical details see: E. Börger, The origins and the development of the ASM method for high level system design and analysis, *J. of Universal Computer Science*, 8(1):2–74, 2002.

[2] E. Riccobene, *Modelli Matematici per Linguaggi Logici*, Ph.D. thesis, University of Catania, Academic year 1991/92.

[3] Y. Gurevich, Evolving Algebras 1993: Lipari Guide, In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

ASM-based modeling, validation and verification techniques into current system engineering methods, truly enriching them (certainly not only rephrasing them in ASM terms), and to identify new challenges for the ASM method.

In fact the invited lectures of ASM 2003 covered not only internal progress and new frontiers for ASMs (lectures by Yuri Gurevich, Microsoft Research, Redmond, USA, and Egon Börger, University of Pisa, Italy), but also some areas of major challenges for new applications of the ASM method:

- object-oriented, component-based design and program verification techniques (lecture by Bertrand Meyer, ETH, Zürich, Switzerland),
- mobile computing (lecture by Gruia-Catalin Roman, Washington University in St. Louis, USA),
- testing (lectures by Antonia Bertolino, ISTI at the Italian Research Council, Pisa, Italy and Klaus Havelund, NASA Research Center, USA),
- concurrency techniques (lecture by Perdita Stevens, University of Edinburgh, UK),
- refinement techniques (lecture by John Derrick, University of Kent, Canterbury, UK).

The contributed *research papers* and *short presentations* of work in progress developed ASM applications and the theory further and contributed to the themes of the invited lectures. The reader will find modeling and analysis work for new kinds of systems (among others for knowledge management, information services, database systems, UML, abstract encryption, the new Java memory model, quantum algorithms), comparative studies of different methods and tools for system description (e.g., big-step and small-step semantics, expression evaluation principles in various programming languages, the MDG tool, ASMs as a platform for the analysis of distributed production control systems at multiple levels of abstraction), advances in the theory of ASMs (analysis of turbo ASMs, decidability problems, support for recursion) and in teaching ASMs, testing, model-checking, etc. In addition ASM 2003 featured industrial experience reports and tool demonstrations.

We thank Uwe Glässer (Simon Fraser University, Vancouver, Canada) and Anatol Slissenko (University 12, Paris, France) for having organized as part of ASM 2003 a round table discussion on the challenging theme of rigorous mathematical models of real time in distributed computing. We also thank the speakers of the round table: Ernst-Rüdiger Olderog (University of Oldenburg, Germany), Andreas Prinz (DResearch Digital Media Systems, Berlin, Germany), and Susanne Graf (Verimag, Grenoble, France). As formulated by Uwe Glässer and Anatol Slissenko, the round table on time in specifications discussed some vision of the following and related topics:

- What are the types of timed systems to consider? What are time constraints and operations over time that arise in practice?
- How are timed systems specified and implemented in practice and what kind of specification languages are demanded or desired by practical specifications?

- How is time represented in existing specification languages? What are the domains of application of these languages and what timed systems are outside these domains?
- Continuous versus discrete time: what are the practical and theoretical advantages or disadvantages of using these models of time?
- What are the theoretical problems in defining the semantics of timed systems?
- How should we represent and simulate time in ASM and ASML?

We thank our colleagues in the program committee and the additional reviewers for their support in evaluating the papers submitted to ASM 2003. We thank Jim Huggins for having placed a copy of the extended abstracts of the work in progress on the ASM website in Michigan. We thank Springer-Verlag, and in particular Alfred Hofmann, for having accepted these proceedings for the LNCS series. We thank our colleagues and students in the University of Catania for their generous help with the organization of ASM 2003. For financial support of the workshop we thank our universities and Microsoft Research. Last, but not least, we thank the participants of ASM 2003 for having made our work useful.

November 2002

Egon Börger
Angelo Gargantini
Elvinia Riccobene

Organization Committee

Egon Börger	(University of Pisa, Italy)
Elvinia Riccobene	(University of Catania, Italy)

Program Committee

Richard Banach	(University of Manchester, UK)
Danièle Beauquier	(University of Paris 12, France)
Egon Börger	(University of Pisa, Italy) (Co-chair)
Jim Davies	(Oxford University, UK)
Gerard R. Renardel de Lavalette	(University of Groningen, The Netherlands)
Uwe Glässer	(Simon Fraser University, Canada)
Peter Päppinghaus	(Siemens, München, Germany)
Elvinia Riccobene	(University of Catania, Italy) (Co-chair)
Robert Stärk	(ETH Zürich, Switzerland)
Peter H. Schmitt	(Universität Karlsruhe, Germany)
Margus Veanes	(Microsoft Research, Redmond, USA)
Wolf Zimmermann	(Universität Halle-Wittenberg, Germany)

Local Committee

Angelo Gargantini	(University of Catania, Italy)
Elvinia Riccobene	(University of Catania, Italy)
Gimpaolo Bella	(University of Catania, Italy)

Additional Reviewers

I. Attali
G. Bella
A. Bertolino
Y. Bryukhov
A. Cavarra
W. Gabrisch
A. Gargantini
Y. Gurevich
W. Hesselink
J. Huggins
J. Jongejan
M. Kardos
W. Löwe
M. Nicolosi Asmundo
P. Scandurra
A. Slissenko
H. Wietze de Haan

Sponsoring Institutions

University of Pisa, Italy
University of Catania, Italy
Microsoft Research, Redmond, WA, USA
Proteo S.p.A., Catania, Italy

Table of Contents

Invited Papers

Software Testing Research and Practice 1
Antonia Bertolino

Abstract State Processes 22
Tommaso Bolognesi and Egon Börger

Recent Advances in Refinement 33
John Derrick and Eerke Boiten

Partial Updates Exploration II 57
Yuri Gurevich and Nikolai Tillmann

Experiments with Test Case Generation and Runtime Analysis 87
Cyrille Artho, Doron Drusinksy, Allen Goldberg, Klaus Havelund, Mike Lowry, Corina Pasareanu, Grigore Roşu, and Willem Visser

A Framework for Proving Contract-Equipped Classes 108
Bertrand Meyer

Mobile UNITY Schemas for Agent Coordination 126
Gruia-Catalin Roman and Jamie Payton

UML and Concurrency 151
Perdita Stevens

Research Papers

A Unified Formal Specification and Analysis of the New Java Memory Models 166
Varsha Awhad and Charles Wallace

Modelling Conditional Knowledge Discovery and Belief Revision by Abstract State Machines 186
Christoph Beierle and Gabriele Kern-Isberner

Formal Description of a Distributed Location Service for Mobile Ad Hoc Networks 204
András Benczúr, Uwe Glässer, and Tamás Lukovszki

Remarks on Turbo ASMs for Functional Equations and Recursion Schemes 218
Egon Börger and Tommaso Bolognesi

Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines 229
Alessandra Cavarra, Elvinia Riccobene, and Patrizia Scandurra

The Hidden Computation Steps of Turbo Abstract State Machines 244
Nicu G. Fruja and Robert F. Stärk

Using Spin to Generate Tests from ASM Specifications 263
Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo

Interfacing ASM with the MDG Tool 278
Amjad Gawanmeh, Sofiène Tahar, and Kirsten Winter

ASMs versus Natural Semantics: A Comparison with New Insights 293
Sabine Glesner

Quantum Computing and Abstract State Machines 309
Erich Grädel and Antje Nowack

Consistent Integration for Sequential Abstract State Machines 324
Marianna Nicolosi Asmundo and Elvinia Riccobene

Deciding the Verification Problem for Abstract State Machines 341
Antje Nowack

An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions 356
Ileana Ober

Privacy, Abstract Encryption and Protocols: An ASM Model – Part I 372
Dean Rosenzweig, Davor Runje, and Neva Slani

A Framework for Modeling the Semantics of Expression Evaluation with Abstract State Machines 391
Wolf Zimmermann and Axel Dold

Extended Abstracts

Using AsmL for Runtime Verification 407
Mike Barnett, Wolfram Schulte, and Nikolai Tillmann

Modeling Information Services on the Basis of ASM Semantics 408
Aleksander Binemann-Zdanowicz and Bernhard Thalheim

Designing the Parlay Call-Control Using ASMs 411
Paolo Falcarin and Alessandra Cavarra

Test Case Generation from AsmL Specifications 413
Wolfgang Grieskamp, Lev Nachmanson, Nikolai Tillmann, and Margus Veanes

Teaching ASMs, Teaching with ASMs: Opportunities in Undergraduate Education 414
James Huggins, Jean Mayo, and Charles Wallace

Using ASM Specifications for Compiler Testing 415
Alexey Kalinov, Alexandre Kossatchev, Alexander Petrenko, Mikhail Posypkin, and Vladimir Shishkov

ASMs as Integration Platform towards Verification and Validation of Distributed Production Control Systems at Multiple Levels of Abstraction 416
Martin Kardos and Ulrich Nickel

AsmL Specification of a Ptolemy II Scheduler 417
Daniel Lázaro Cuadrado, Peter Koch, and Anders P. Ravn

ASM Specification of Database Systems 418
Andreas Prinz and Bernhard Thalheim

The Computable Kernel of ASM 421
Wolfgang Reisig

A Non-standard Approach to Operational Semantics for Timed Systems 423
Heinrich Rust

Parallelism versus Nondeterminism: On the Semantics of Abstract State Machines 425
Wolfram Schulte

Author Index 427

Software Testing Research and Practice

Antonia Bertolino

ISTI-CNR, Area della Ricerca CNR di Pisa, Italy
bertolino@iei.pi.cnr.it

Abstract. The paper attempts to provide a comprehensive view of the field of software testing. The objective is to put all the relevant issues into a unified context, although admittedly the overview is biased towards my own research and expertise. In view of the vastness of the field, for each topic problems and approaches are only briefly tackled, with appropriate references provided to dive into them. I do not mean to give here a complete survey of software testing. Rather I intend to show how an unwieldy mix of theoretical and technical problems challenge software testers, and that a large gap exists between the state of the art and of the practice.

1 Foreword

It is with pleasure that I received the invitation to present my views and perspectives on Software Testing at the 10th International Workshop on Abstract State Machines. The testing of software is the research field I have been involved in for several years now, and in this talk I will argue the important role of testing in research and in practice.

On the other hand, I will not hide a little trepidation to talk in front of the somewhat exclusive ASM "club", that has been enrolling highly qualified researchers and practitioners from worldwide for almost 20 years [18]. There is an underlying motivation: whereby the ASM approach to system analysis and design preaches very precise abstract models, and a stepwise and throughout mathematically verifiable refinement from these abstract models down to executable code, in contrast testing is intrinsically an empirical, pragmatic activity.

Testing by its nature can never conclude anything mathematically valid, as it amounts to taking a sample and trying to infer a generally valid judgement on the whole from the observed part. To complicate things, when the object of testing includes software, in making the inference we cannot rely on any certain continuity property [44] as in the testing of physical systems. We can/must work towards making the sampling less ad hoc, and more systematic. We can/must develop support tools to automate the clerical activities, and to partially automate the intellectual tasks. We can/must try to incorporate quantitative, measurable notions within the analysis of test results. Yet, the fact remains that *program testing can be used to show the presence of bugs, but never to show their absence*, as incisively stated by Dijkstra [28] as far as thirty years ago.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 1–21, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Though, testing is a challenging activity, and can greatly contribute to the engineering of quality programs. I contrasted the formal rigor of the ASM approach to the inherent pragmatism of testing. However, the contrast is only artificial: on the contrary, it is just within a well formalized process, such as ASM's, that testing provides the highest benefits. Indeed, by looking at the constant concern on putting ASMs to use in industrial applications [18], and at efforts devoted to exploit ASM models to drive testing [42, 43], I feel confident that this talk will find in the ASM community positive reception and a fertile soil.

1.1 Paper Structure

I introduce a definition of software testing and related terminology in Sect. 2. I discuss test case selection in Sect. 3 and other relevant issues in Sect. 4. I overview testing in relation with the development process in Sect. 5. Very brief conclusions are drawn in Sect. 6.

2 Software Testing

To start with I will re-propose here the definition of Software Testing introduced in [9][1]. That definition was worked out as a reference framework allowing for both arranging the relevant issues of software testing within a unifying vision, and tracing the most active research directions.

Definition 1. *Software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the specified expected behavior.*

The underlined terms identify each a corresponding key concern for software testers. Let me briefly analyze them:

- dynamic: this term is meant to explicitly signify that testing implies executing the program on (valued) inputs. To be precise, the input value alone is not always sufficient to determine a test, as a system behavior generally depends on the system state (and non deterministic system might even react with different behaviors to a same input in the same state). Different from testing, and complementary with it, are static analysis techniques, such as peer review and inspection [33, 31] (sometimes improperly referred to as "static testing"); program execution on symbolic inputs, or symbolic evaluation [24]; formal verification techniques, such as theorem provers, model checkers, data-flow analyzers. Yet another practice is to apply traditional test approaches to "test" the executable specifications: this might be useful to

[1] This reference is a chapter within the Guide to the Software Engineering Body of Knowledge (SWEBOK), providing an overview and references to acquire the basic "generally accepted" notions behind the Software Testing Knowledge Area, and could serve as a starting point for testing newcomers.

early reveal potential inconsistencies in the model. All these approaches are important, but are left outside the scope of this paper. I focus here expressly on testing the implementation. Indeed, while all the above approaches are useful to evaluate the internal "correctness" of a software system, testing is the only procedure allowing for the assessment of its behavior when executed within the target environment.

- finite: for even simple programs, so many test cases are theoretically possible that exhaustive testing (i.e., exercising it on every input) would require even years to execute (e.g., Dijkstra [28] calculated that the exhaustive testing of a multiplier of two 27-bit integers taking "only" -at the time- some tens of microseconds for a single multiplication would require more than 10000 years). Generally, the whole test set can be considered infinite. In contrast, the number of executions that can realistically be observed must obviously be finite (and affordable). Clearly, "enough" testing to get reasonable assurance of acceptable behavior should be performed. This basic need points to well known problems of testing, both technical in nature (criteria for deciding test adequacy) and managerial in nature (estimating the effort to put in testing). Testing always implies a trade-off between limited resources and schedules, and inherently unlimited test requirements;
- selected: the many proposed *test criteria* essentially differ in how they select the (finite) test suite. Testers should be constantly aware that different techniques may yield largely different effect, also depending on the context (kind of application, maturity of the process and the organization, expertise of testers, tool support are among the influencing factors). How to identify the most suitable selection criterion under given conditions is a very complex problem [76]; in practice risk analysis techniques and test engineering expertise are applied;
- expected: it must be possible (although not always easy) to decide whether the observed outcomes of program execution are acceptable or not, otherwise the testing would be useless. The observed behavior may be checked against user's expectations (commonly referred to as *testing for validation*) or against a specification (*testing for verification*, also said conformance testing). The test pass/fail decision is commonly referred in the testing literature to as the *oracle problem*.

2.1 Fault vs. Failure

Although strictly related, fault and failure denote different notions. When a test is executed and the oracle response is "fail", this means that the program exposed an undesired behavior: properly this is called a *failure*, and can be defined [55] as the deviation of the delivered service from the function the program was intended for.

Instead, the originating cause of the failure is said a *fault*. A fault may remain dormant long time, until it is activated and brings the program to a state which, if propagated to the observable output, can lead to a failure: this intermediate

unstable state is indicated as an *error*. Hence, the chain

$$\text{fault} \rightarrow \text{error} \rightarrow \text{failure}$$

expresses a causality relationship, which can be iterated recursively, as a fault itself can be seen as the consequence of a failure of some other system: for instance a coding fault is a failure of the program developer.

Mechanisms of how faults propagate to failures have been extensively modelled [69, 60, 77] and empirically studied [34, 50], and even specific test techniques have been devised based on the resulting models, such as for example the weak mutation approach [51].

Notwithstanding its intuitive meaning, defining formally a fault is extremely difficult. If, as it is often done, a fault is characterized for practical purposes as "what must be changed" (in code, specifications, design, ...) to eliminate the failed behavior, we should then accept the fact that widely different fixes could be done to remove a same failure. Therefore, the term fault is ambiguous and should be avoided: in [37], the authors suggest to refer instead to "failure regions" of the input domain such that when a point within them is executed the program fails. Having said this, I will anyway refer to "fault" in the following, when it is necessary to explicitly distinguish between the effect (the failure) and its cause.

Important considerations apply for testers. By testing we can expose failures, but only subsequent off-line analysis can identify the originating faults. Such analysis in modern large system can be extremely costly, and some techniques for automating fault localization in code have been developed, e.g. [22].

Another important difference regards the evaluation of a system through the test results: counting failures or counting faults may yield quite different results[2] and implications. A historical reference in this regard is [1], in which empirical data showed that most faults are ineffective, as they would never be activated in the lifetime of a system: probably only ten per cent of design faults are worth fixing. Although quite obvious, this consideration is ignored in several studies that base their evaluation of a test selection criterion not on a measure of the potential impact of failures found (as done in [37]), but just on their number.

3 Test Selection

The problem of test cases selection has been the largely dominating topic in software testing research to the extent that "software testing" is often taken as a synonymous for "test case selection". Researchers creativity seems endless in proposing novel criteria for picking out a "good" sample of potential program behaviors. A comprehensive survey until 1997 is [87], but new criteria have been proposed since.

An important point to always keep in mind is that what makes a test a "good" one does not have a unique answer, but it changes depending on the context, on

[2] Note that the number of faults can be significantly lower from the number of failures exposed by the tests, as a same fault can stimulate several failures, if the program is not fixed immediately.

the specific application, and on the goal for testing. The most common interpretation for "good" would be "able to detect many failures"; but again precision would require to specify what kind of failures, as it is well known and experimentally observed that different test criteria trigger different types of faults [6, 86]. Therefore, it is always preferable to spend the test budget to apply a combination of diverse techniques than concentrating it on just one, even if shown the most effective [57].

Moreover, fault removal is only one of the two potential goals for software testing; the second one is evaluating some specified quality, e.g., reliability testing, usability testing, performance testing, and so on. Indeed, the final result of a validation process should be to achieve an objective measure of the confidence that can be put on the software being developed. Test suites that are good for one purpose, could not be as good for another.

In general, a test criterion is a means of deciding which a "good" set of test cases should be. A more precise definition is provided below.

Definition 2. *A test criterion C is a decision predicate defined on triples (P, RM, T), where P is a program, RM is a reference model related to P, and T is a test suite. When $C(P, RM, T)$ holds, it is said that T satisfies criterion C for P and RM.*

This definition refers to a broad category of test approaches referred to as "partition" testing: the adopted reference model RM induces (directly or indirectly) a partitioning of the program input domain into subdomains and it is tacitly assumed (this is the underlying test hypothesis) that for every point within a same subdomain the program either succeeds or fails. Therefore, thanks to the test hypothesis only one or few points within each subdomain need to be checked, and this is what allows for getting a finite set of tests out of the infinite domain. The basic notion of grouping the inputs into subdomains that constitute equivalence classes for test purposes were early formalized in [85, 68].

Partition testing is contrasted with "random testing", by which instead test inputs are randomly drawn from the whole domain according to a specified distribution[3](most often the uniform or the operational distributions). The relative merits of these two different test philosophies have been highly debated (see e.g. the sequence of [30, 45, 84, 37]).

Testers can use a test criterion for guiding in proactive way the selection of test cases (so that when the selection terminates the criterion is automatically fulfilled), or for checking after the fact if the executed (and anyhow else selected) suite is sufficient. In the latter case, the criterion provides a *stopping rule* for testing, i.e., for given P and RM a test suite T satisfying $C(P, RM, T)$ is deemed to be *adequate*. For instance, a tester could execute a test suite manually derived from the analysis of the requirements specification document, and use a coverage

[3] An approach somewhere in the middle between pure partition testing and random testing is the statistical testing approach [74], by which the distribution of the inputs is designed so that the randomly drawn test cases have a high probability to satisfy a specified criterion.

analyzer tool during test execution for measuring the percentage of program branches covered, stopping the testing as soon as this percentage reaches a fixed threshold.

Test criteria can be classified according to the kind of *RM* [9]: it can be as informal as "tester expertise", or strictly formal, as in the case of conformance testing from a formal specification or also of code-coverage criteria. The advantages of a formal *RM* are evident: the selection of test cases, or otherwise adequacy evaluation, can be automatized.

Many are the factors of relevance when a test selection criterion has to be chosen. A framework within which existing test criteria can be categorized and compared with regard to their potential utilization in a context has been developed and empirically evaluated [76].

Paradoxically, test case selection seems to be on the other hand the least interesting problem for test practitioners. A demonstration of this low interest is the paucity of commercial automated tools for helping test selection and test input generation, in comparison with a profusion of support tools for handling test execution and re-execution (or regression test) and for test documentation, both in the specification of tests and in the logging and analysis of test results. Indeed, much progress could be done in this direction, as test criteria that have been state-of-art for more than twenty years, and could greatly improve test cost/effectiveness, remain still almost unknown to practitioners (as recently discussed in [8]). The most practiced test selection criterion in industry probably is still tester's expertise, and indeed expert testers may perform as very good selection "mechanisms". Empirical investigations [6] showed in fact that tester's skill is the factor that mostly affect test effectiveness in finding failures.

3.1 Selection Criteria Based on Code

Code-based testing has been the dominating trend in software testing research during the late 70's and the 80's. One reason is certainly that in those years in which formal approaches to specification were much less mature and pursued than now, the only *RM* formalized enough to hopefully allow for the automation of test selection or adequacy measurement was the code.

These criteria are also called path-based, because they map each test input to a unique path p on the flowgraph corresponding to *RM*. The ideal and yet unreachable target of code-based testing would be the exhaustive coverage of all possible paths along the program control-flow. The basic test hypothesis here is that by executing a path once, potential faults related to it will be revealed, i.e., it is assumed that every input executing a same path will either fail or succeed (which is not necessarily true, of course).

Full path coverage is not applicable, because banally every program with unbounded loops would yield an infinite number of paths. Even limiting the number of iterations within program loops, usually practised tactic in testing, the number of tests would remain infeasibly high. Therefore, all the proposed code-based criteria attempt to realize cost/effective approximations to path coverage, by identifying specific (control-flow or data-flow) elements of a program that are

deemed to be relevant for revealing failures, and by requiring that enough paths to cover all such elements be executed. In particular, in data flow-based testing, the graph model is annotated with information about how the program variables are defined and used, and a test is aimed at exercising how a value assigned to a variable is used along different control-flow paths.

The landmark paper in code-based testing is [67], in which a family of criteria is introduced, based on both control-flow and data-flow. A *subsumption* hierarchy between the criteria was derived, based on the inclusion relation such that a test suite satisfying the subsuming criterion is guaranteed to also satisfy the (transitively) subsumed criterion. This family hierarchy has remained still today an important reference point: whenever a new criterion is proposed, its place in this hierarchy is located. Indeed, since 1985, many other criteria have been added to the Rapps-Weyuker family (e.g., [29, 35]).

To automate path-based testing, the program is modelled as a graph and the entry-exit paths over the graph are considered. Finding a set of graph paths fulfilling a coverage criterion thus becomes a matter of properly visiting the graph (see for instance [12]). However, two though problems immediately arise:

- not all graph paths correspond to actual program paths, i.e., the applied graph visiting algorithm could select some paths which are not executable. The *unfeasible paths* problem is a crux of code-based test automation, and its incidence grows as we go up in the subsumption hierarchy.
- finding an input that executes a selected graph path is not only an undecidable problem in principle [82], but also a quite difficult one to solve in practice. Traditionally, symbolic execution [23] was attempted, with scarce practical success. More recent approaches include dynamic generation based on optimization [54] or genetic algorithms [65].

In consequence of these problems, code-based criteria should be more properly used as adequacy criteria. After all, code-based test selection is a tautology: it looks for potential problems in a program by using the program itself as a reference model. In this way, for instance, faults of missing functionalities could never be found. It is hence more sensible to use another artifact as the *RM* and measure code coverage while tests are executed, so to evaluate the thoroughness of the test suite. If some elements of the code remain uncovered, additional tests to exercise them should be found, as it can be a signal that the tests do not address some function that is coded. A warning is worth that "exercised" and "tested" are not synonymous, and as shown in [35] under most existing code-based criteria even 100% coverage could even leave untested statements.

3.2 Selection Criteria Based on Specifications

In specification-based testing, the *RM* is derived in general from the documentation relative to program specifications. Depending on how these are expressed, largely different techniques are possible. Very early approaches [62] looked at the Input/Output relation of the program "black-box" and manually derived equivalence classes, or boundary conditions, or cause-effect graphs.

Lot of researchers have tried to automatize the derivation of test cases from formal or semiformal specifications. Early attempts include algebraic specifications [7], VDM [27], and Z [48], while a more recent collection of approaches can be found in [49]. An influential work is [64], introducing the Category-Partition (CP) method for the automated generation of functional tests from annotated semi-formal specifications. CP is a simple, intuitive, yet effective method for functional testing, and since its appearance its basic principle has been applied to specifications in several languages, including recently UML [5].

Also in specification based testing a graph model is often derived and some coverage criterion is applied on this model. A number of methods rely on coverage of specifications modelled as a Finite State Machine (FSM). A review of these approaches is given in [17].

LTS-based conformance testing has been the subject of extensive research [20]: given the LTS for the specification S and one of its possible implementation I, expressed using LTSs or Input/Output Transition Systems (IOTSs), some "Implementation relations" (imp) are defined, formalizing the notion of conformance of I with respect to S [75]. In such a formal setting, test generation algorithms must produce sound test suites, i.e., such that models passing the test correspond to conformant implementations. A related approach [53] uses Input/Output LTS (IOLTS) to formalize the specification, the implementation and the test purposes (TP). An approach for the automatic, on-the-fly generation of test cases to verify the TP has been implemented in the Test Generation and Verification (TGV) [73] tool.

As expectable, specification-based testing nowadays focuses on testing from UML models. A spectrum of approaches is being developed, ranging from strictly formal testing approaches based on UML statecharts [56], to approaches trying to overcome UML limitations requiring OCL additional annotations [19], to pragmatic approaches using the design documentation as is and proposing automated support tools [5]. The currently ongoing Agedis [2] project aims at developing a UML based test tool, based on the already cited TGV environment.

3.3 Other Criteria

Several other test criteria have been proposed, but size limitations do not allow me to tackle them in detail. I only cite that the *RM* could be given by expected or hypothesized faults, such as in Error guessing [62], or Mutation testing [26]. Another criterion is operational testing [59], in which *RM* is how the users will exercise a system (see also Section 4.4).

4 Selecting the Test Cases is Not the Only Issue

Having derived a test suite is only one piece of the testing puzzle and more complex pieces have to be put in place before the picture is complete. Other test related activities present technical and conceptual difficulties that are underrepresented in research (but well present to practitioners): the ability to launch

the selected tests (in a controlled host environment, or worse in the tight target environment of an embedded system); deciding whether the test outcome is acceptable or not; if not, evaluating the impact of the failure and finding its direct cause (the fault) and indirect one (Root Cause Analysis); judging whether the test campaign is sufficient, which in turn would require having at hand effectiveness measures of the tests: one by one all these tasks present though challenges.

These activities have received marginal attention in software testing research. One argument is that being these issues technological in kind, in contrast with the more theoretical and intuitive problem of test selection, the approaches pursued are specific to an application context, and are less easily generalizable.

4.1 Test Execution

Forcing the execution of the test cases (manually or automatically) derived according to one of the criteria above mentioned might not be obvious. As said, if a code-based criterion is followed, it provides us with entry-exit paths over the graph model, and test inputs that execute the corresponding program paths need be found. If a specification-based criterion relying on coverage of a LTS or a FSM is adopted, the test cases are paths over them corresponding to sequences of events, that are specified at the abstraction level of the specifications; more precisely they are labels within the signature of the adopted language. To derive concrete test cases, the labels of the specification language must be translated into corresponding labels at code level, and eventually into execution statements to be launched on the GUI of the used test tool.

The larger the system, the more desirable to keep the abstraction level of the specification high enough to preserve an intuitive view of relevant functions on which the system architect can reason. The translation difficulty also depends on the degree of formality of the refinement process from the high level specification to design and to code. In fact, if a strict formal refinement process is adopted, the translation can be as simple as applying some substitution rules (coded along development). In real world practice, this is not generally the case, and a heavy effort for manual translation may be required, as discussed in [11].

In addition to translating the specified test cases into executable runs, another requirement is the ability to put the system into a state from which the specified tests can be launched. This is sometimes referred to as the *test precondition*. In synchronous systems before a specific command can be executed, several runs in sequence are required to put the system in the suitable test precondition. An effective way to deal with this is to arrange the selected test cases into suitable sequences, such that each test leaves the system into a state that is the precondition for the subsequent test in the sequence. The problem has been early formalized and tackled (for VDM specifications) in [27]. This approach cannot easily scale up to the integration testing of large, complex systems, in which the specified tests involve actions specific to exercise a subsystem. It can be alleviated by always defining the tests at the external interfaces, i.e., as complete I/O sequences.

A new dimension to the problem is added in concurrent systems allowing for nondeterminism. In this case, the system behavior not only depends on the internal status, but also on the interleaving of events with system tasks and other concurrently running systems. When testing reveals a failure, the task of recreating the conditions that made it occur is termed test *replay*. Exact replay requires mechanisms for capturing the happening of synchronization events and memory access, and for forcing the same order of events when a test is replayed. The deterministic approach was originally introduced in [21] for the Ada language. More recent algorithms and tools have been proposed for Java [32]. The approach is highly intrusive, as it requires to heavily instrument the system. A more pragmatic approach otherwise is to keep repeating a test until the desired sequence is observed (fixing a maximum number of iterations).

An orthogonal problem arises during integration testing, when testing only parts of a larger system. Indeed, the testing task itself requires a large programming effort: to be able to test a piece of a large system we need to "simulate" the surrounding environment of the piece under tests (e.g., the caller and called methods). This is done by developing ad hoc drivers and stubs; some commercial test tools exist than can facilitate these tasks.

4.2 Test Oracles

An important component of testing is the oracle. Indeed, a test is meaningful only if it is possible to decide about its outcome. The difficulties inherent to this task, often oversimplified, had been early articulated in [83].

Ideally, an oracle is any (human or mechanical) agent that decides whether the program behaved correctly on a given test. The oracle is specified to output a *reject* verdict if it observes a failure (or even an error, for smarter oracles), and *approve* otherwise. Not always the oracle can reach a decision: in these cases the test output is classified as inconclusive.

Different approaches can be taken. In a scenario in which a limited number of test cases is executed, sometimes even derived manually, the oracle is the tester himself/herself, who can either inspect a posterior the test log, or even decide a priori, during test planning, the conditions that make a test successful and code these conditions into the employed test driver.

When the tests cases are automatically derived, and when their number is quite high, in the order of thousands, or millions, a manual log inspection or codification is not thinkable. Automated oracles must then be implemented. But, of course, if we had available a mechanism that knows in advance and infallibly the correct results, it would not be necessary to develop the system: we could use the oracle instead! Hence the need of approximate solutions.

In some cases, the oracle can be an earlier version of the system that we are going to replace with the one under test. A particular instance of this situation is regression testing, in which the test outcome is compared with earlier version executions (which however in turn had to be judged correct). Generally speaking, an oracle is derived from a specification of the expected behavior. Thus,

in principle, automated derivation of test cases from specifications have the advantage that by this same task we get an abstract oracle specification as well. However, the gap between the abstract level of specifications and the concrete level of executed tests only allows for partial oracles implementations, i.e., only necessary (but not sufficient) conditions for correctness can be derived.

In view of these considerations, it should be evident that the oracle might not always judge correctly. So the notion of *coverage* [4] of an oracle is introduced. It could be measured by the probability that the oracle rejects a test (on an input chosen at random from a given probability distribution of inputs), given that it should reject it [15].

A recent survey of approaches to automated test oracles is [4]. The authors overview the use of assertions embedded into the program and providing run-time checking capability; the use of conditions expressly specified to be used as test oracles, in contrast with using the "pure" specifications (i.e., written to model the system behavior and not for run-time checking); and finally the analysis of execution traces.

4.3 Analysis of Test Results

Starting to talk of test case selection, I mentioned that researchers continuously strive for finding "good" criteria. But what makes a criterion better than another? In general terms a notion of *effectiveness* must be associated with a test case or an entire test suite, but test effectiveness does not yield a universal interpretation. Some people misconceive the meaning of coverage measures and confuse coverage with effectiveness. More properly, coverage is relative to the tests themselves and measure their thoroughness in exercising RM. Being systematic and trying of not leaving elements of code or of the specification untested is certain a prudent norm, but should be properly understood for what it is. A real measure of test effectiveness should be relative to the program and should allow testers to quantify the effect of testing on the program's attribute of interest, so that the test process can be kept under control.

We have already mentioned that one intuitive and diffuse practice is to count the number of failures or faults detected. The test criterion that founds the highest number could be deemed the most useful. Even this measure has drawbacks: as tests are gathered and more and more faults are removed, what can we infer about the resulting quality of the tested program? for instance, if we continue testing and no new faults are found for a while, what does this imply? that the program is "correct", or that the tests are ineffective?

The most objective measure is a statistical one: if the executed tests can be taken as a representative sample of program behavior, than we can make a statistical prediction of what would happen for the next tests, should we continue to use the program in the same way. This reasoning is at the basis of software reliability.

[4] It is just an unfortunate coincidence the usage with a quite different meaning of the same term adopted for test criteria.

Documentation and analysis of test results require discipline and effort, but form an important resource of a company for product maintenance and for improving future projects.

4.4 The Notion of Software Reliability

Reliability is one of the several factors, or "ilities", into which the quality of software has been decomposed to be able to measure it, and by far it is the factor for which today the most developed theory exists. Strictly speaking, software reliability is the probability that the software will execute without failure in a given environment for a given period of time [59]. In particular, to assess the reliability "in a given environment", this input distribution should approximate as closely as possible the *operational distribution* for that environment.

The extreme difficulty of identifying an operational distribution for software systems is one of the arguments brought by opponents of software reliability [45]. However, the practical approach proposed by Musa to define even a course operational profile by grouping different typologies of users and functionalities has demonstrated great effectiveness ([59] Chapt. 5).

After the tests have been executed, a reliability measure is obtained by analyzing the test results against a selected reliability model. *Reliability growth* models are used to model the situation that as a program is tested and failures are encountered, the responsible faults are found and fixed, and the program is again submitted to testing; this process continues until it is evaluated that an adequate reliability measure has been reached. These models are in fact called reliability growth models, understanding that as the software undergoes testing and is repaired, its reliability increases.

The first software reliability growth models appeared in the early 1970's. Today, tens of models are in use (see [59], Chapt. 3). No "universally best" model can be individuated; studies have shown that although a few models can be rejected because they perform badly in all situations, no single model always performs well for all potential users. However, at the current state of the art, it is usually possible to obtain reasonably accurate reliability predictions.

Reliability, or statistical, models are used instead after the debugging process is finished to assess the reliability one can expect from the software under test, which is also called *life testing*. If a set n of test cases randomly drawn from the *operational* distribution is executed, and f is the number of test inputs that have failed, then an estimate of reliability is very simply given by

$$R = 1 - \frac{f}{n}$$

When there are not enough observed failures against which the predicted reliability can be checked, we could evaluate the probability that the reliability is high enough by using a purely statistical estimate. Assuming that the probability of failure per input ϑ is constant over time, and that each test outcome is independent from the others, an upper bound for the failure probability can be estimated with confidence (1-C), where:

$$C \leq (1 - \vartheta)^n$$

In safety critical applications the reliability requirements are so high that no feasible test campaign can ever be sufficient to demonstrate they have been reached. This is known as the "ultra-high reliability" problem, and only the combined usage of different means for getting evidence of correct behavior can have some hope to attack the problem [58].

5 Putting It altogether in a Seamless Process

While several problems have been discussed so far, the real big challenge ahead is to work out a unified process within which all these test tasks are gracefully complementing each other, and testing as a whole is not an activity detached from construction (and posterior to it), but the two things, building and checking, become two faces of the same coin, two seamlessly integrated activities. What does this mean? For instance, that there is not anymore in the Laboratory a CASE tool from vendor X for modelling and design using methodology A, and another tool from vendor J for testing according to technique B, but a *unique* environment for modelling, design *and* testing, relying on the same notation, terminology, and reference framework. As obvious as this sounds, reality is yet quite far from it.

Detachment between development and testing has been the key cause for leaving testing as a second-class activity, something that is recognized as important but that anyway one can get rid of in emergency. Researchers have their responsibility in this, as software testing research has not liaised with research in formal methods and modelling notations, and results itself fragmented into specialized sectors. A notable example in this sense is the separation between the two communities addressing the test of general applications, and the more specialized test of communication systems and protocols. Until very recently research in these two sectors has been carried out with almost no interactions, but an interesting initiative is now attempting to coordinate researchers throughout Europe into the Testnet Network [72].

Another evidence is given by the UML: while a big emphasis has been devoted to how to use UML for modelling and how the notation expressiveness and precision could be improved, relatively little attention has been focused on how UML could advance testing.

5.1 (Specify/Design for) Testability

What are the properties that enhance the capability to test a program? How should we model/specify/design a system during development so that testing: i) requires less effort, and ii) is more effective? Clearly, practical answers to such questions would have a great impact in software industry, as testing continues to be very expensive. The term "testability" captures this intuitive notion of "ease" to test a program and is included in the ISO/IEC 9126 [52] list of software quality attributes.

Research has addressed the two questions i) and ii) above along two separate tracks. For example, in [3] the authors define a notion of "testability" linked to the effort needed to accomplish the coverage required by various testing methods, e.g., branch coverage. In [38] the testability of a component is evaluated through the I/O mapping structure and the fan-in/fan-out measure. This deterministic notion of testability attempts to base the evaluation of the effort to be put in testing on measurable properties of program, with the aim to suggest design approaches that can eventually make testing less expensive. This is useful work towards addressing question i), but this measure of testability has no necessary relationship to the ability to detect faults (which was question ii).

A different, probabilistic notion of testability is that in [78, 15]. Program "testability" is here taken as the probability that a program will fail under test, if it is faulty. Having a knowledge of the testability of a program in this sense would permit to interpret with higher precision the results of testing and obtain more favorable predictions than allowed by "black-box" based inference alone.

5.2 Test Phases

Testing of large systems is organized into phases, i.e., the testing task is partitioned into a phased process, addressing at each step the testing of a subsystem. Integration testing refers to the testing of the interactions between subsystems along system composition. An incremental systematic approach should be taken, as opposed to a big-bang approach. The aim is to keep complexity under control and to eventually arrive at the final stage of system testing with all the composing subsystems extensively tested. At each stage test selection is closely related with the object under test.

Some white-box approaches propose to derive integration test cases based on the call structure among modules and measure inter-procedural coverage [47]. In OO systems, integration tests consist of interleaved sequences of module paths and messages, and they are derived considering the interaction patterns between objects, for instance the collaborations or the client-server hierarchy [16].

Until very recently, specification- or design-based integration testing has been performed ad hoc. With the emergence of Software Architecture (SA) as a discipline to describe and analyze large, complex systems, several authors have advocated the use of architectural models also to drive testing, and in particular to select relevant behaviors of interactions between system components. The key issue would be to use the SA *not only* to derive a set of test cases, *but also* to drive the decomposition of a system into subsystems, thus identifying a suitable integration test process. There is not much work on this. In [10, 11], we derive from a formal SA model the LTS, on which graph coverage techniques can be applied. The added value of working at the architectural level of description is that the tester can single out relevant sequences by operating a (trace-preserving) abstraction (called an *observation-function*) over the complete LTS and thus deriving an abstract ALTS. By applying different observation-functions, different system configurations can be tested.

It should be emphasized that most practical testing actually corresponds to *regression testing*, i.e., after software modification or evolution already a set of already passed tests are re-executed to verify that no new faults have been introduced. As regression testing might be very expensive, lot of research has been devoted to select an optimal subset of tests [70].

5.3 Test Patterns

A practical instrument to the design of complex systems are patterns. A design pattern is an abstract description of a recurring problem, together with a general arrangement of elements and procedures that has proved to be useful to solve it. Patterns have always been used by expert designers and engineers: they form their cultural expertise. The famous book from Gamma et al. [39] was the first methodical attempt to document and catalogue design patterns for OO design.

Symmetric to design patterns comes the idea of identifying and logging interesting and recurrent patterns in the testing of complex systems. Unfortunately, there is not much work in this sense. The encyclopedic Binder's book [16] collects patterns for test design in the context of OO systems, organized into class test patterns, subsystem test patterns and system test patterns, with each pattern offering a stand-alone solution for a scope-specific test plan. Certainly more research and empirical work towards the definition of test patterns is desirable, and an eventual catalogue would testify a more mature status of the discipline of testing.

5.4 Testing in Component-Oriented Development

The emerging paradigm of development is Component Based (CB), in which a complex system is built by assembling simpler pieces, similarly to what is routine practice in any traditional engineering domain. Ideally, by adopting a CB approach, production times and costs can be greatly reduced, while at the same time more manageable and reliable systems can be obtained. However, the real take-up of a CB approach requires to solve a lot of still open questions (see [25] for a possible list).

Certainly a revision of development processes is necessary, and this revision also involves the testing stage. In particular [13], the traditional three phases of the testing process (unit, integration and system test) can be revisited in terms of three corresponding new phases, respectively referred to as component, deployment and system test. Indeed, the tests performed by the developer are clearly inadequate to guarantee a dependable usage of the component in the target application environment (that of the system constructor). The testing should be repeated also in the final environment, while the component interacts with the other components.

Component testability [40] has been hence identified as an important quality to make easier the evaluation of components by means of testing. Four different perspectives on testability are identified:

- Component Observability: this defines the ease with which the behaviour of a component can be observed in term of its in/out parameters
- Component Traceability: this defines the capability to trace its behavior and the state of its attributes
- Component Controllability: this defines the ease with which the behaviour of the component can be controlled on its in/out, operations and behaviour
- Component Understandability: this refers to how much component information is provided and how well it is presented

In particular, to increase the Understandability of a component, some authors have proposed to augment the components themselves with additional information, in form of meta-data [63, 71].

A new challenge that arises from the inherently distributed nature of CB production is the so-called *component trust problem*, which refers to the difficulty to understand what a component really does. The trust problem is strongly related to the customer capability to validate the adequacy of a candidate component to his/her specific purposes. Different approaches are under study; in particular some authors support the constitution of independent agencies that act as a software certification laboratory (SCL), performing extensive testing in different usage domains [79, 80]. Another approach proposes a form of self certification, in which the component developer releases to the component user a set of test cases, in the form of a XML file, to retest the component in the final application environment [61].

Regarding testing, in our knowledge there is not much work addressing the problem. A first approach proposes to embed test cases in the component itself (Built-In Tests) in the form of methods externally visible and specifically assigned to the execution of tests [81]. A disadvantage of this approach is the size growth of components. To overcome this problem, another approach introduced the concept of a testable architecture. This architecture foresees that the components implement a particular interface for testing purposes, that permits to execute pre-built tests without the necessity to include them in the component code [41].

In this context, we are currently developing a framework for facilitating the deployment testing of components [14]. In this framework the specification of functional tests by the system assembler and the implementation details of components are decoupled thanks to the use of generic *test wrappers.* These dynamically adapt the component interfaces, permitting to easily reconfigure the subsystem under test after the introduction/removal/substitution of a component, and to optimize test reuse, by keeping trace of the test cases that directly affect the wrapped component.

6 A Brief Conclusive Remark

I hope this general overview has showed the several complex facets of the often undervalued testing task, and has moved the curiosity of some ASM scientists while challenging them to put in place this visionary dream of a seamless

build&test process within ASM. About the references provided, they span a period of over than 20 years. Not always the citations are the most recent entries for a topic, but sometimes have been chosen as the most representative. A recent look to future research perspectives is provided in [46].

Although there is much room for automation in each of the involved activities, the tester's expertise remains essential as much as a need for approximate solutions under constrains remains. Again, testing is a challenging and important activity. After having opened with the famous Dijkstra aphorism that highlights its limitations, let me conclude with the as famous quotation from Knuth[5]: *Beware of bugs in the above code; I have only proved it correct, not tried it.*

References

[1] Adams, E.: Optimizing Preventive Service of Software Products. IBM J. of Research and Development **28**,1 (1984) 2–14 4

[2] Agedis: Automated Generation and Execution of Test Suites for DIstributed Component-based Software. <http://http://www.agedis.de/index.shtml> 8

[3] Bache, R., Müllerburg, M.: Measures of Testability as a Basis for Quality Assurance. Software Engineering Journal. **5** (March 1990) 86–92 14

[4] Baresi, L., Young, M.: Test Oracles. Tech. Report CIS-TR-01-02. On line at <http://www.cs.uoregon.edu/ michal/pubs/oracles.html> 11

[5] Basanieri, F., Bertolino, A., Marchetti, E.: The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects. Proc. 5th Int. Conf. UML 2002, Dresden, Germany. LNCS 2460 (2002) 383–397 8

[6] Basili, V. R., Selby, R. W.: Comparing the Effectiveness of Software Testing Strategies. IEEE Trans. Software Eng. **13**(12) (1987) 1278–1296 5, 6

[7] Bernot, G., Gaudel M. C., Marre, B.: Software Testing Based On Formal Specifications: a Theory and a Tool. Software Eng. Journal **6** (1991) 387–405 8

[8] Bertolino, A.: ISSTA 2002 Panel: Is ISSTA Research Relevant to Industrial Users?. In [36] 201-202 (see also following entries 203–209) 6

[9] Bertolino, A.: Knowledge Area Description of Software Testing. Guide to the SWEBOK, Joint IEEE-ACM Software Engineering Coordinating Committee. (2001) On-line at: <http://www.swebok.org> 2, 6

[10] Bertolino, A., Corradini, F, Inverardi, P., Muccini, H.: Deriving Test Plans from Architectural Descriptions. Proc. Int. ACM Conf. on Soft. Eng. (2000) 220–229 14

[11] Bertolino, A., Inverardi, P., Muccini, H.: An Explorative Journey from Architectural Tests Definition downto Code Tests Execution. Proc. Int. Conf. on Soft. Eng. (2001) 211–220 9, 14

[12] Bertolino, A., Marré, M.: A General Path Generation Algorithm for Coverage Testing. Proc. 10th Int. Soft. Quality Week, San Francisco, Ca. (1997) pap. 2T1 7

[13] Bertolino, A., Polini, A.: Re-thinking the Development Process of Component-Based Software, ECBS 2002 Workshop on CBSE, Lund, Sweden (2002) 15

[14] Bertolino, A., Polini, A.: WCT: a Wrapper for Component Testing. Proc. Fidji'2002, Luxembourg (to appear) (2002) 16

[5] This sentence was in a personal communication to P. van Emde Boas (1977).

[15] Bertolino, A., Strigini, L.: On the Use of Testability Measures for Dependability Assessment IEEE Trans. Software Eng. **22**(2) (1996) 97–108 11, 14
[16] Binder, R. V.: Testing Object-Oriented Systems - Models, Patterns, and Tools. Addison-Wesley (2000) 14, 15
[17] Bochmann, G. V., Petrenko, A.: Protocol Testing: Review of Methods and Relevance for Software Testing. Proc. Int. Symp. on Soft. Testing and Analysis (ISSTA), Seattle (1994) 109–124 8
[18] Börger, E.: The Origins and the Development of the ASM Method for High Level System Design and Analysis. J. of Universal Computer Science **8**(1) (2002) 2–74. On line at: <http://www.jucs.org/jucs_8_1/the_origins_and_the> 1, 2
[19] Briand, L., Labiche, Y.: A UML-Based Approach to System Testing. Software and Systems Modeling **1**(1) (2002) 10–42 8
[20] Brinksma, E., Tretmans, J.: Testing Transition Systems: An Annotated Bibliography. Proc. of MOVEP'2k, Nantes (2000) 44–50 8
[21] Carver, R. H., Tai, K.-C.: Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. IEEE Trans. on Soft. Eng. **24**(6) (1998) 471–490 10
[22] Choi, J-D., Zeller, A.: Isolating Failure-Inducing Thread Schedules. In [36] 210–220 4
[23] Coward, P. D.: Symbolic Execution Systems - A Review. Software Eng. J. (1988) 229–239 7
[24] Clarke, L. A., Richardson, D. J.: Applications of Symbolic Evaluation. The J. of Systems and Software **5** (1985) 15–35 2
[25] Crnkovic, I.: Component-based Software Engineering - New Challenges in Software Development. John Wiley&Sons (2001) 15
[26] DeMillo, R. A., Offutt, A. J.: Constraint-Based Test Data Generation. IEEE Trans. Software Eng. **17**(9) (1991) 900–910 8
[27] Dick, J., Faivre, A.: Automating The Generation and Sequencing of Test Cases From Model-Based Specifications. Proc. FME'93, LNCS 670 (1993) 268–284 8, 9
[28] Dijkstra, E. W.: Notes on Structured Programming. T. H. Rep. 70-WSK03 (1970) On line at: <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF> 1, 3
[29] Duesterwald, E., Gupta, R., Soffa, M. L.: Rigorous Data Flow Testing Through Output Influences. Proc. 2nd Irvine Software Symposium, Irvine, CA. (1992) 131–145 7
[30] Duran, J. W., Ntafos, S. C.: An Evaluation of Random Testing. IEEE Trans. Software Eng. **SE-10**(4) (1984) 438-444 5
[31] Dunsmore, A., Roper, M., Wood, M.: Further Investigations into the Development and Evaluation of Reading Techniques for Object-Oriented Code Inspection. Proc. 24th Int. Conf. on Soft. Eng. Orlando, FL, USA (2002) 47–57 2
[32] Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded Java Program Test Generation. IBM Systems Journal **41** (2002) 111–125 10
[33] Fagan, M. R.: Design and Code Inspections to Reduce Errors in Program Development. IBM Systems Journal **15**(3) (1976) 182–211 2
[34] Fenton, N. E., Ohlsson, N.: Quantitative Analysis of Faults and Failures in a Complex Software System. IEEE Trans. Software Eng. **26**(8) (2000) 797-814 4
[35] Forgacs, I., Bertolino, A.: Preventing Untestedness in Data-flow Based Testing. Soft. Testing, Verification and Reliability **12**(1)(2001) 29–61 7

[36] Frankl, P. G. (Ed.): Proc. ACM Sigsoft Int. Symposium on Soft. Testing and Analysis ISSTA 2002, Soft. Engineering Notes **27**(4) Roma, Italy (July 2002) 17, 18, 19
[37] Frankl, P. G., Hamlet, R. G., Littlewood, B., Strigini, L.: Evaluating Testing Methods by Delivered Reliability. IEEE Trans. Software Eng. **24**(8) (1998) 586–601 4, 5
[38] Freedman, R. S.: Testability of Software Components. IEEE Trans. Software Engineering **17**(6) (1991) 553–564 14
[39] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley (1994) 15
[40] Gao, J.: Component Testability and Component Testing Challenges, Proc. ICSE Workshop on Component-Based Soft. Eng. (2000) http://www.sei.cmu.edu/cbs/cbse2000/paper/ 15
[41] Gao, J., Gupta, K., Gupta, S., Shim, S.: On Building Testable Software Components. Proc. ICCBSS2002, LNCS 2255 (2002) 108–121 16
[42] Gargantini, A., Riccobene, E.: ASM-Based Testing: Coverage Criteria and Automatic Test Sequence. J. of Universal Computer Science **7**(11) (2001) 1050–1067. On line at: <http://www.jucs.org/jucs_7_11/asm_based_testing_coverage> 2
[43] Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating Finite State Machines from Abstract State Machines. In [36] 112-122 2
[44] Hamlet, D.: Continuity in Software Systems. In [36] 196-200 1
[45] Hamlet, D., Taylor, R.: Partition Testing Does Not Inspire Confidence. IEEE Trans. Software Eng. **16**(12) (1990) 1402-1411 5, 12
[46] Harrold, M. J.: Testing: A Roadmap. In A. Finkelstein (Ed.), The Future of Software Engineering, ACM Press (2000) 63–72 17
[47] Harrold, M. J., Soffa, M. L.: Selection of Data for Integration Testing.IEEE Software (March 1991) 58–65 14
[48] Hierons, R. M.: Testing from a Z Specification. Soft. Testing, Verification and Reliability **7**(1997) 19–33 8
[49] Hierons, R., Derrick, J. (Eds): Special Issue on Specification-based Testing. Soft. Testing, Verification and Reliability **10** (2000) 8
[50] Hiller, M., Jhumka, A., Suri, N.: PROPANE: An Environment for Examining the Propagation of Errors in Software. In [36] 81–85 4
[51] Howden, W. E.: Weak Mutation Testing and Completeness of Test Sets. IEEE Trans. Software Eng. **8**(4) (1982) 371–379 4
[52] ISO/IEC 9126, Information Technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use. (1991) 13
[53] Fernandez, J.-C., Jard, C., Jeron, T., Nedelka, L., Viho, C.: Using On-the-fly Verification Techniques for the Generation of Test Suites. Proc. of the 8th Int. Conf. on Computer Aided Verification (1996) 8
[54] Korel, B.: Automated Software Test Data Generation. IEEE Trans. Software Eng. **16**(8) (1990) 870–879 7
[55] Laprie, J. C.: Dependability - Its Attributes, Impairments and Means. In [66] 3–18 3
[56] Latella, D., Massink, M.: On Testing and Conformance Relations for UML Statechart Diagrams Behaviours. In [36] 144–153 8
[57] Littlewood, B., Popov, P. T., Strigini, L., Shryane, N.: Modeling the Effects of Combining Diverse Software Fault Detection Techniques. IEEE Trans. Software Eng. **26**(12) (2000) 1157–1167 5
[58] Littlewood, B., Strigini, L.: Validation of Ultra-High Dependability for Software-based Systems. Communications of the ACM. **36**(11) (1993) 69–80 13

[59] Lyu, M. R. (Ed.): Handbook of Software Reliability Engineering. IEEE Comp. Soc. Press/McGraw-Hill (1996) 8, 12
[60] Morell, L. J.: A Theory of Fault-based Testing. IEEE Trans. Software Eng. **16**(8) (1990) 844–857 4
[61] Morris, J., Lee, G., Parker, K., Bundell, G. A., Lam, C. P.: Software Component Certification. IEEE Computer (September 2001) 30–36 16
[62] Myers, G. J.: The Art of Software Testing. Wiley. (1979) 7, 8
[63] Orso, A., Harrold, M. J., Rosenblum, D.: Component Metadata for Software Engineering Tasks. Proc. EDO2000, LNCS 1999 (2000) 129–144 16
[64] Ostrand, T. J., Balcer, M. J.: The Category-Partition Method for Specifying and Generating Functional Tests. ACM Comm. **31**(6) (1988) 676–686 8
[65] Pargas, R., Harrold, M. J., Peck, R.: Test-Data Generation Using Genetic Algorithms. J. of Soft. Testing, Verifications, and Reliability **9** (1999) 263–282 7
[66] Randell, B., Laprie, J. C., Kopetz, H., Littlewood B., Eds.: Predictably Dependable Computing Systems, Springer (1995) 19, 20
[67] Rapps, S., Weyuker, E. J.: Selecting Software Test Data Using Data Flow Information. IEEE Trans. Software Eng., **SE-11** (1985) 367–375 7
[68] Richardson, D.J, Clarke, L. A.: Partition Analysis: A Method Combining Testing and Verification. IEEE Trans. Software Eng., **SE-11** (1985) 1477–1490 5
[69] Richardson, D., Thompson, M. C.: The Relay Model for Error Detection and its Application. Proc. 2nd Wksp Soft. Testing, Verification, and Analysis. Banff Alberta, ACM/Sigsoft and IEEE (July 1988) 223–230 4
[70] Rothermel, G., Harrold, M. J.: Analyzing Regression Test Selection Techniques. IEEE Trans. Software Eng., **22**(8) (1996) 529–551 15
[71] Stafford, J. A., Wolf, A. L.: Annotating Components to Support Component-Based Static Analyses of Software Systems, Proc. Grace Hopper Celeb. of Women in Computing (2001) 16
[72] TESTNET – Integration of Testing Methodologies. <http://http://www-lor.int-evry.fr/testnet/> 13
[73] TGV–Test Generation from transitions systems using Verification techniques. On line at <http://www.inrialpes.fr/vasy/cadp/man/tgv.html> 8
[74] Thevenod-Fosse P., Waeselynck H., Crouzet Y.: Software Statistical Testing. In [66] 253–272 5
[75] J. Tretmans. Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation. Computer Networks and ISDN Systems **29** (1996) 49–79 8
[76] Vegas, S.: Characterisation Schema for Selecting Software Testing Techniques. Int. Software Engineering Research Network (ISERN'01) Strachclyde, Scotland (2001) 3, 6
[77] Voas, J. M.: PIE: A Dynamic Failure-Based Technique. IEEE Trans. Software Eng. **18**(8) (1992) 717-727 4
[78] Voas, J. M., Miller, K. W.: Software Testability: The New Verification. IEEE Software (May 1995) 17–28 14
[79] Voas, J.: Certifying Off-the-Shelf Software Components. IEEE Computer (June 1998) 53–59 16
[80] Voas, J.: Developing a Usage-Based Software Certification Process. IEEE Computer (August 2000) 32–37 16
[81] Wang, Y., King, G., Wickburg, H.: A Method for Built-in Tests in Component-based Software Maintenance, Proc. European Conference on Soft. Maintenance and Reengineering (1998) 186–189 16

[82] Weyuker, E. J.: Translatability and Decidability Questions for Restricted Classes of Program Schemas. SIAM J. on Computers **8**(4) (1979) 587–598 7

[83] Weyuker, E. J.: On Testing Non-testable Programs. The Computer Journal **25**(4) (1982) 465–470 10

[84] Weyuker, E. J., Jeng, B.: Analyzing Partition Testing Strategies. IEEE Trans. Software Eng. **17**(7) (1991) 703–711 5

[85] Weyuker, E. J., Ostrand, T. J.: Theories of Program Testing and the Application of Revealing Subdomains. IEEE Trans. Software Eng. **SE-6**(1980) 236-246 5

[86] Wood, M., Roper, M., Brooks, A., Miller, J.: Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study. Proc. ESEC/FSE, LNCS 1301 (1997) 262–277 5

[87] Zhu, H., Hall, P. A. V., May, J. H. R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys, **29** (1997) 366–427 4

Abstract State Processes

Tommaso Bolognesi[1] and Egon Börger[2]

[1] CNR, ISTI, Pisa
bolognesi@ei.pi.cnr.it
[2] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. Process-algebraic languages offer a rich set of structuring techniques and concurrency patterns which allow one to decompose complex systems into concurrently interacting simpler component processes. They abstract however almost entirely from a notion of system state. The method of Abstract State Machines (ASMs) offers powerful abstraction and refinement techniques for specifying system dynamics based upon a most general notion of structured state. The evolutions of the state are governed however by a fixed and typically unstructured program, called 'rule', which describes a set of abstract updates occurring simultaneously at each step (synchronous parallelism). We propose to incorporate into one machine concept the advantages offered by both structuring techniques, and introduce to this purpose Abstract State Processes (ASPs), i.e. evolving processes (extended ASM programs which are structured and evolve like process-algebraic behaviour expressions) operating on evolving abstract states the way traditional ASM rules do.

1 Introduction

This work has been motivated by the observation that both the Process Algebra approach and the Abstract State Machine approach to system specification—one based on evolving behaviour and the other on evolving state ('evolving algebra')—offer each its own advantages in terms of natural expressiveness and flexible support for design and analysis when applied to the proper aspects of the system under investigation. The prospects for a fruitful coupling of the two approaches seem to be good.

For foundational as well as for historical reasons (see [7] for the details) basic ASMs as they originally appeared in [14] are characterized by a single-agent, synchronous, parallel execution model where the program is fixed and unstructured, namely coming as a set of simultaneously executed rules of form

if *Condition* **then** *Updates*

with a set *Updates* of simultaneously executed function updates $f(s_1, \ldots, s_n) := t$. A large number of successful applications surveyed in [4, 5] worked with this model of computation which, extended by the **forall** construct, is also at the basis of the recently obtained proof for the synchronous parallel case of the

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 22–32, 2003.
© Springer-Verlag Berlin Heidelberg 2003

ASM thesis from a small number of postulates [2]. Basic ASMs offer besides their synchronous atomic actions in a global state no other execution control structure, in particular no sequencing, no iteration, no submachine call concept. The same phenomenon can be observed for Abrial's notion of Abstract Machines [1] which come as non-executable pseudo-code without sequencing or loop.

In contrast, process algebra descriptions of systems are focussed on the evolution of structured programs, called behaviour expressions, which are built up from subexpressions in a compositional way to support modular specifications and structured proof methods. A process algebraic specification describes the event patterns an observer can experience at the interface of the described system, as well as the interactions among system components, namely as sequence of behaviour expressions $B_0, B_1, \ldots$ where B_i is transformed into B_{i+1} by the occurrence of an atomic internal or an observable (inter-)action. The variety of expressive means for the decomposition of systems into simpler component processes which concurrently communicate and interact contrasts with the absence of a notion of structured state (except for instantiations of process parameters).

We explore here the idea of specifying concurrent systems by Abstract State Processes (ASPs), i.e. evolving processes whose 'steps' make abstract states evolve the way basic ASM rules do. Since stepping through the program of an ASP becomes part of the overall system evolution, the single steps of a sequential execution, of an iteration and of a submachine execution become visible. This white-box view differs from the black-box view of these concepts which has been realized by turbo ASMs in [9] for an integration of standard programming constructs, including recursion [8], into the synchronous parallelism of basic ASMs, and has been successfully applied in [21] for modelling and analysing the semantics of Java and its implementation on the Java Virtual Machine. The white-box view we define here naturally leads to equip ASPs with two more constructs:

- *interleaving*, which allows parallel processes to act independently from each other, one at a time,
- *selective synchronization* (see [3]), which allows interleaved processes to act in parallel if and only if they all 'share a selected event' (read: contribute to a selected update).

Our definitions are tailored not to involve asynchronous ASMs[1]. With the synchronous form of parallelism which is characteristic for basic or turbo ASMs, the difference betweeen mono-agent and multi-agent systems does not constitute a semantical difference in the underlying computation model; although it matters if agents are used to define disjoint name spaces for a separation of state components. From that point of view interleaving represents a scheduling principle which allows one to view multiple-agent processes as executed by one agent. Therefore the Abstract State Processes we are going to define naturally lead to mono-agent machines. We hope that the integration of interleaving and selective synchronization into the synchronous parallelism of ASMs will provide

a useful guideline for extending current implementations of turbo ASMs to an implementation of asynchronous multi-agent ASMs.

The program evolution of an ASP could be described as a walk through the abstract syntax tree of the behaviour expression, following the scheme used for modeling the execution of Java programs in [21]. Instead we adopt the structured programming approach to make the process-algebraic constructs for communication and synchronization of processes stand out explicitly. In doing so we are not primarily concerned with the minimality and orthogonality requirements which have guided the definition of many process-algebraic systems as well as of basic ASMs. Our choice of constructs is mostly led by the pragmatic needs of system and software engineering applications, which require a balance between conceptual simplicity and manipulative ease.

In this paper we provide only the basic definitions. A further analysis has to be postponed to a future occasion. In the last section we formulate some questions we consider worth to be investigated. We suppose the reader to have at least some elementary knowledge of the fundamental intuitions of process algebra (see [16, 18, 19, 20]) and of ASMs (see [14]). For a succinct definition of the extension of basic ASMs to ASPs we use the notational framework of the ASM book [10].

2 Semantics of Standard Abstract State Processes

We define here the extension of basic ASMs by process-algebraic concepts of evolving programs (usually called behaviour expressions) providing, in addition to basic ASM features, white-box sequencing and submachine execution. In the next section we add to this interleaving and selective synchronization.

We preserve the state transformation mechanism of basic ASMs whereby to define the semantics of a construct it suffices to define the update set U it yields in an arbitrary state $\mathfrak{A}$. To this we add the definition of the remaining (called the *residual*) program P the given construct leaves for further execution. Thus we describe for each ASP construct P which pair (P', U) of residual program P' and update set U it yields in an arbitrary state $\mathfrak{A}$. We do this following the inductive definition of the syntax of ASPs, providing for each ASP expression P one or two inference rules which define how to derive statements of form $\text{yields}(P, \mathfrak{A}, P', U)$. We concentrate our attention here upon the definition of the semantics of ASPs, given that the extension of the syntax of ASMs to that of ASPs is a routine matter.

For a better overview we put the definition into two tables. Table 1 is adapted from [9, 10] for the constructs defining turbo ASMs, namely **skip**, assignment,

[1] Passing from multi-agent synchronous to asynchronous ASMs adds considerable complexity to the notion of runs, defined at the end of [15] under the name of 'distributed' runs. On the theoretical side, no natural postulates are known to derive for asynchronous ASMs an analogon to the synchronous parallel ASM thesis of [2]. We therefore advice to use asynchronous ASMs only where really appropriate. See in this context also [8].

Table 1. Inductive definition of the semantics of standard ASP rules

$\overline{\text{yields}(\textbf{skip}, \mathfrak{A}, \textbf{nil}, \emptyset)}$	
$\overline{\text{yields}(f(s_1, \ldots, s_n) := t, \mathfrak{A}, \textbf{nil}, \{((f, (\llbracket s_1 \rrbracket^{\mathfrak{A}}, \ldots, \llbracket s_n \rrbracket^{\mathfrak{A}})), \llbracket t \rrbracket^{\mathfrak{A}})\})}$	
$\dfrac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, P', U)}$	if $\llbracket \varphi \rrbracket^{\mathfrak{A}} = \mathit{true}$
$\dfrac{\text{yields}(Q, \mathfrak{A}, Q', V)}{\text{yields}(\textbf{if } \varphi \textbf{ then } P \textbf{ else } Q, \mathfrak{A}, Q', V)}$	if $\llbracket \varphi \rrbracket^{\mathfrak{A}} = \mathit{false}$
$\dfrac{\text{yields}(P\frac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U)}{\text{yields}(\textbf{let } x = t \textbf{ in } P, \mathfrak{A}, P', U \cup \{(x', a)\})}$	$a = \llbracket t \rrbracket^{\mathfrak{A}}$ x' fresh
$\dfrac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for each } a \in I = \mathit{range}(x, \varphi, \mathfrak{A})}{\text{yields}(\textbf{par } \{P(x) \mid \varphi(x)\}, \mathfrak{A}, \textbf{par } \{P_a \mid a \in I\}, \bigcup_{a \in I} U_a \cup \{(y_a, a)\})}$	y_a fresh
$\dfrac{\text{yields}(P\frac{x'}{x}, \mathfrak{A}[x' \mapsto a], P', U) \quad \text{for some } a \in I = \mathit{range}(x, \varphi, \mathfrak{A})}{\text{yields}(\textbf{choose } \{P(x) \mid \varphi(x)\}, \mathfrak{A}, P', U \cup \{(x', a)\})}$	x' fresh
$\dfrac{\text{yields}(P, \mathfrak{A}, P', U)}{\text{yields}(P \textbf{ then } Q, \mathfrak{A}, P' \textbf{ then } Q, U)}$	if $P' \neq \textbf{nil}$
$\dfrac{\text{yields}(P, \mathfrak{A}, \textbf{nil}, U)}{\text{yields}(P \textbf{ then } Q, \mathfrak{A}, Q, U)}$	
$\dfrac{\text{yields}(P\frac{t_1 \cdots t_n}{x_1 \cdots x_n}, \mathfrak{A}, P', U)}{\text{yields}(r(t_1, \ldots, t_n), \mathfrak{A}, P', U)}$	$r(x_1, \ldots, x_n) = P$ rule declaration

conditional expressions, **let**, **par**, **choose**, **forall**, sequencing and parameterized submachine call. In adapting the definition of turbo ASMs to ASPs, besides integrating the program evolution we replace the black-box view of sequential execution and submachine calls defined in [9] by the corresponding white-box ASP constructs. For white-box sequencing we write **then** instead of **seq**.

The execution of **skip** and assignment processes yields the empty residual process **nil**. A conditional process as defined in Table 1 blocks other processes with which it may be synchronized if in case the condition is true its subprocess P, or otherwise Q, cannot proceed. Therefore one has to distinguish two interpretations of **if** *Cond* **then** R. The *persistent* one is defined as **if** *Cond* **then** R **else nil**; it has a blocking effect in case *Cond* is false since no inference rule is provided to execute the empty program **nil**. The *transient* version is defined as **if** *Cond* **then** R **else skip** which in case *Cond* is false uses the inference rule for **skip**. We use the persistent version as the default. One may compare this with the blocking evaluation of guards e.g. in the high-level design language COLD [12], whereas in ASMs the rule of a 'blocked' process does not prevent other rules from being executed in parallel.

For the formulation of processes which involve the manipulation of logical variables it is notationally convenient to view states $\mathfrak{A}$—structures with finitely

many domains, functions and relations—as sets of pairs (l, v) of locations l and their values v. Locations are pairs (f, a) of a function name f (say of arity n) and a sequence $a = (a_1, \ldots, a_n)$ of arguments in the domain of f. We identify free variables with 0-ary function symbols (parameterless locations) so that their interpretation is incorporated into $\mathfrak{A}$[2]. Thus we write $\mathfrak{A}[x' \mapsto a]$ for the extension of $\mathfrak{A}$ by the new location x' with value a.

In the **let**-construct local variables appear to which precomputed values are assigned. In basic or turbo ASMs this preliminary computation step remains implicit as part of the unique computation step associated to the machine **let** $x = t$ **in** P, whose execution implies a form of sequentialization which is typical for the call-by-value discipline, namely to first compute t in the given state and then to execute P with the computed value recorded in the local variable x. These variables are called logical because they are not updated and their binding to the current value of t holds only for the atomic execution of P. Since in ASPs such a program P is not executed atomically, but may lead after one step to a residual program P' which involves further steps, the incarnation x' of the variable x with its interpreation by the value t has to survive until the residual process has become empty (reduced to **nil**). This holds analogously also for the other ASP constructors.

We therefore use for each execution of a constructor $c(x)\,P$ in a given state a fresh instance of x, say x', standing for a new 0-ary function which records for the entire execution of the constructor body P the value assigned in the given state to the parameter x. Formally this makes the signature of ASPs dynamic, though the dynamics is restricted to creating new incarnations of local variables, whereas in traditional ASMs the signature is static. By imposing the condition of x' being fresh (meaning by this that it is sufficiently new not to be mixed up with any other variable[3], namely that is not used before and not simultaneously anywhere else) we guarantee that the currently determined value for x will not collide with any other value determined in a different process or in a different step for the same parameter x ('same' in the syntactic sense). For brevity we write x, although we allow it to denote a tuple of parameters.

For notational uniformity we write **par** $\{P(x) \mid \varphi(x)\}$ instead of **forall** x **with** φ **do** P. In case φ evaluates to finitely many elements, **par** $\{P_1, \ldots, P_n\}$ stands for **par** $\{P(x) \mid \varphi(x)\}$ where $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$. This avoids having to fuss with fixed parameter instances. Similarly for **choose** $\{P(x) \mid$

[2] In [9, 10] this interpretation is indicated by a separate environment function ζ, following a widespread notational practice of mathematical logic.

[3] See [10, Ch.2] for a simple ASM characterization of this use of the **import**-construct applied to the reserve set. That characterization also covers our usage of fresh local variables x', avoiding to introduce a new inference rule into the semantical description of ASPs. One can use a similar expedient to the one introduced in [8] to separate the 'logical' (read-only) variables from the variables for locations (write-variables). It suffices to write *varIncar*(x') instead of x' with a monadic function *varIncar* which takes variable incarnations as arguments; to guarantee that different incarnations of x are stored in different locations it suffices to pass to *varIncar* different arguments x', x''.

Table 2. Semantics of interleaving and selective synchronization

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \text{for some } a \in I = \mathit{range}(x, \varphi, \mathfrak{A})}{\text{yields}(\mathbf{intlea}\ \{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{intlea}\ \{P_b \mid b \in I\}, U_a \cup \bigcup_{b\in I}\{(y_b, b)\})}$$

where y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \mathit{Loc}(t)^{\mathfrak{A}} \in \mathit{Loc}(U_a) \text{ for each } a \in I}{\text{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_a \mid a \in I\}, \bigcup_{a\in I} U_a \cup \{(y_a, a)\})}$$

where I=$\mathit{range}(x, \varphi, \mathfrak{A})$, y_a fresh for all $a \in I$

$$\frac{\text{yields}(P\frac{y_a}{x}, \mathfrak{A}[y_a \mapsto a], P_a, U_a) \quad \mathit{Loc}(t)^{\mathfrak{A}} \notin \mathit{Loc}(U_a) \text{ for some } a \in I}{\text{yields}(\mathbf{sync}\ (t)\{P(x) \mid \varphi(x)\}, \mathfrak{A}, \mathbf{sync}\ (t)\{P_b \mid b \in I\}, U_a \cup \bigcup_{b\in I}\{(y_b, b)\})}$$

where $I = \mathit{range}(x, \varphi, \mathfrak{A})$, y_c fresh for all $c \in I$, $P_b = P\frac{y_b}{x}$ for $b \in I \setminus \{a\}$

$\varphi(x)\}$ and for the synchronization and interleaving operators **sync** (t), **intlea** defined in the next section. See Table 3 for a summary of some alternative notations borrowed from the process-algebraic literature. When applying ASP operators to sets of processes, it is notationally convenient to assume the following implicit transformation which will not be mentioned furthermore: whenever the application of an inference rule leads to a residual program $\mathit{oper}\{P\}$, where *oper* is any of the operators and $\{P\}$ is a singleton set of processes, the residual program is considered as automatically rewritten into P. Similarly, $\mathit{oper}\{\}$ is rewritten into **nil**. By $\varphi\frac{t}{x}$ we denote the result of replacing all free occurrences of the variable x in φ by the term t. By $\mathit{range}(x, \varphi, \mathfrak{A})$ of a formula φ with distinguished variable x in a state $\mathfrak{A}$ we denote the set of all elements a of $\mathfrak{A}$ that make the formula true if x is interpreted by a.

choose-processes as defined in Table 1 are blocking in case there is nothing to choose from, namely when the choice set is empty, since for this case we provide no inference rule. This is in contrast to the ASM **choose**-construct which in case of an empty choice set is usually treated as equivalent to **skip**.

The submachine call defined in Table 1 is by reference. The call-by-value version can be defined using the **let**-construct.

3 ASPs for Interleaving and Selective Synchronization

We add here to standard ASPs the interleaving operator **intlea** and the related selective synchronization operator **sync** (t). Their semantics is defined in Table 2. Interleaving is a choice among processes to perform the next step where however the programs of the not-chosen processes remain in force for subsequent choices. To exhibit the analogy of interleaving to synchronous parallelism, which involves a form of universal quantification, we formulate the **intlea**-rule for an arbitrary set of processes determined by a property $\varphi(x)$. As a consequence, to

Table 3. Syntactic variations of some ASP constructs

choose x **with** φ **do** P $\mid \{P(x) \mid \varphi(x)\}$	**choose** $\{P(x) \mid \varphi(x)\}$
forall x **with** φ **do** P $\mid\mid \{P(x) \mid \varphi(x)\}$	**par** $\{P(x) \mid \varphi(x)\}$
$\mid t \mid \{P(x) \mid \varphi(x)\}$	**sync** $(t)\{P(x) \mid \varphi(x)\}$
$\mid\mid\mid \{P(x) \mid \varphi(x)\}$	**intlea** $\{P(x) \mid \varphi(x)\}$
operator$\{P_1, \ldots, P_n\}$ (**operator** $=\mid, \mid\mid, \mid t \mid, \mid\mid\mid$)	**operator**$\{P(x) \mid \varphi(x)\}$ **where** $\{P(x) \mid \varphi(x)\} = \{P_1, \ldots, P_n\}$

determine the set of the processes P_b which have not been chosen but are put into interleaving with the residual process P_a of the one process chosen for execution, one needs to keep track of the instantiations of $P(x)$ by the elements b which satisfy the condition φ. Formally this comes up to bind pairwise different new incarnations y_b of the parameter x to b. In the case of an explicitly given set $P_1, \ldots, P_n$ of processes where the parameter instances are fixed, our notational convention eliminates the need to reinstantiate these fixed parameter indeces $1 \leq i \leq n$.

The synchronization operator **sync** (t) allows one to prevent the occurrence of actions which do not involve an update of (the location determined by) t by all the synchronized processes, similar to the *restriction* operation in SCCS. We write for the set of locations appearing in an update set $Loc(U) = \{loc \mid (loc, v) \in U \text{ for some } v\}$. The set of locations determined by a set T of terms in a state is denoted by $Loc(T)^{\mathfrak{A}} = \{(f, (\llbracket t_1 \rrbracket^{\mathfrak{A}}, \ldots, \llbracket t_n \rrbracket^{\mathfrak{A}})) \mid f(t_1, \ldots, t_n) \in T\}$. We write $Loc(t)^{\mathfrak{A}}$ for $Loc(\{t\})^{\mathfrak{A}}$.

Whereas every ASM program yields in every state an update set (though it may be inconsistent in which case the computation is abrupted), when executing an ASP it may happen that the current program in the current state has no yield because no axiom or inference rule can be applied. This produces a form of ASP termination which does not exist for ASMs, that of a (static) deadlock. If it occurrs it might represent a bug in the modelled system. It can arise in the following cases for the residual program:

- A conditional expression with blocked argument process.
- A sequential expression with blocked first argument.
- A choice expression where all alternatives are blocked.
- A parallel expression with at least one blocked process.
- An interleaving expression where all component processes are blocked.

4 Two Examples

We illustrate by two simple examples how the operators **intlea**, **sync** (t) allow one to make standard scheduling and handshaking disciplines transparent.
Local sequential scheduling of subprocesses. Using basic ASMs one can express a general modular scheme for scheduling the execution of rules belonging to a set $\mathfrak{R}$ which may even be dynamic. It suffices to update by a *scheduler* a function *select* which chooses the rule $R(select)$ to be executed next.

$\textsc{Scheduling}(\mathfrak{R}, scheduler) =$
  par $\{R(select), select := scheduler(\mathfrak{R}, select)\}$

An instance of the scheme is to require the *scheduler* to make *select* behave as interleaving. Another example is the Round Robin principle which is defined by the equation $scheduler(select) = select + 1 \bmod n$ for any number n of processes. The use of such a *scheduler* function, which may also be dynamic, allows one to restrict the mere interleaving by conditions which may still leave some freedom to choose the next rule to be executed, maybe also in dependence of the state where *scheduler* is used.

However, to describe with basic ASMs the combination of interleaving with white-box sequential execution would lead to a programming solution which seems to be less natural than using the ASP operators **intlea** and **then**. Imagine for example the set $\mathfrak{R}$ consists of processes R of form

$\mathrm{R} = First_R$ **then** $Second_R$

which are constrained to be interleaved such that the order of execution of the subprocesses $Second_R$ is determined by the order of execution of the subprocesses $First_R$. For a local realization of such a scheduling we equip each process **self** with its instance **self** $.ticket$ of a location *ticket* which keeps track of the order in which the interleaving operator chooses the subprocesses $First_R$ for execution. This allows one to a) locally record by a copy of the current *ticket* value 'when' **self** has been called to start the execution of its first subprocess, and b) to locally advance *ticket* to the next free position in the dynamic ordering of calls of subprocesses $First_R$. This means to place two updates to get and advance the current ticket in parallel with $First_{\mathbf{self}}$: **self** $.ticket := ticket$ and $ticket := ticket + 1$. Then $Second_{\mathbf{self}}$ can be scheduled when its ticket is say 'displayed', i.e. when the guard **self** $.ticket = display$ is true, adding to $Second_{\mathbf{self}}$ an update to advance the *display*. This yields the following transformation of **intlea** $(\mathfrak{R})$ into **intlea** $(\textsc{LocalSeqSched}(\mathfrak{R}))$.

$\textsc{LocalSeqSchedule}(R) =$ $First'_R$ **then** $Second'_R$ **where**
  $First'_R =$**par** $\{First_R, \textsc{GetAndAdvanceOrderPos}(R)\}$
  $\textsc{GetAndAdvanceOrderPos}(R) =$
    par $\{R.ticket := ticket, ticket := ticket + 1\}$
  $Second'_R =$ **if** $displayed(R.ticket)$ **then**
    par $\{Second_R, \textsc{AdvanceDisplay}(R)\}$

$displayed(Ticket) \;=\; (Ticket = display)$
$\textsc{AdvanceDisplay}(R) = (display := display + 1)$

Handshaking. Process communication via rendez-vous (handshaking) is often specified via so-called *gates* at which the participating agents have to 'agree on offered values'. This can be viewed as a special case of shared memory communication, namely via gate locations g shared for reading and/or writing. The general scheme for two processes P, Q is as follows, where we use predicates φ, ψ to determine the choice the processes may have for agreeing upon a consistent update of gate g with a value determined by terms s, t:

$\textsc{Handshaking}(P, \varphi, s, Q, \psi, t) = \mathbf{sync}\ (g)\{R, S\}\ \mathbf{where}$
$R = \mathbf{choose}\ x\ \mathbf{with}\ \varphi(x)\ \mathbf{in}\ (g := s(x)\ \mathbf{then}\ P(x))$
$S = \mathbf{choose}\ y\ \mathbf{with}\ \psi(y)\ \mathbf{in}\ (g := t(y)\ \mathbf{then}\ Q(y))$

This ASP formulation of handshaking realizes the 'agreement on values offered at a gate' by the consistency condition for the gate updates. If one wants to faithfully reflect also that in the process-algebraic view, gates are only virtually updated, serving only as communication medium, one can declare the updates of g to be transient, i.e. not relevant for the resulting state transformation. This can be done by simply not considering these transient updates in the definition of the next-state function, which associates new states to given states and update sets. A similar expedient has been used already in [9] to restrict update sets to updates of not-local functions and for defining the error handling construct. The technique can be exploited further for defining practical hiding disciplines for ASMs and ASPs.

5 Related and Further Work

The relation of our work to the rich process algebra and ASM literature can easily be traced through [11] and [6]. The related approaches to combine a state based method, namely Z, Object-Z or B, with behavioural concepts from process algebraic systems, namely CSP and CCS, are extensively discussed in Chapter 18 of [11], and the references to the relevant papers can be found there. The main difference of our approach with respect to previous work is that we allow, as states, arbitrary structures (as in Z) and as behaviour expressions arbitrary machines (not only single operations, executed one per time, as in Z or B), thus gaining from process algebras the richness of behavioural structure without loosing the generality of the state-based ASM method (see the ASM book [10]). As a result, the general abstraction and refinement techniques survive which made complex real-life applications of ASMs successful (see [7] for a survey).

An interesting start for the further analysis of ASPs is to characterize the possible runs of ASPs, similar to the elegant characterization of turbo ASM runs in [13]. Another direction of study is to investigate to what extent one can adapt to ASPs the structured and proof-oriented process-algebra refinement techniques in [11]. Maybe this leads to a fruitful integration of these techniques into the practical instantiations of the general refinement scheme which has been formulated

for ASMs in [6]. A third field of interest is to investigate which process-algebraic proof rules can be generalized to support deductive and possibly mechanically verified reasoning about ASP runs.[4] We would also like to see the impact of ASPs on the various existing implementations of basic or turbo ASMs, in particular for implementing asynchronous ASMs. Last but not least we are looking for interesting real-life practical applications of ASPs.

Acknowledgement

We thank unknown referees for critical remarks upon a draft of this paper.

References

[1] J.-R. Abrial. *The B-Book*. Cambridge University Press, 1996. 23
[2] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002. 23, 24
[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1987. 23
[4] E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer-Verlag, 1995. 22
[5] E. Börger. High level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999. 22
[6] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14:?–?, 2002. 30, 31
[7] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002. 22, 30
[8] E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, LNCS. Springer-Verlag, 2003. 23, 24, 26
[9] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000. 23, 24, 25, 26, 30
[10] E. Börger and R. Stärk. *Abstract State Machines. A method for high-level system design and analysis*. Springer, 2003. 24, 26, 30
[11] J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Springer, 2001. 30

[4] In reaction to the announcement of T. Bolognesi's talk on this work at MSR in Redmond on November 15, 2003, Leslie Lamport sent us an unpublished manuscript [17] where he develops a notation for TLA actions and predicates which reflect process-algebraic concepts, aiming at exploiting process-algebraic rules for reasoning about programs in a programming language defined by operators for sequential composition, choice, interleaving and iteration with a break statement.

[12] L. M. G. Feijs and H. B. M. Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992. 25

[13] N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*, LNCS. Springer-Verlag, 2003. 30

[14] Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991. 22, 24

[15] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. 24

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 24

[17] L. Lamport. Adding "Process Algebra" to TLA. January 1995. 31

[18] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989. 24

[19] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998. 24

[20] S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach*. Wiley, 2000. 24

[21] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. 23, 24

Recent Advances in Refinement

John Derrick and Eerke Boiten

Computing Laboratory, University of Kent, Canterbury, CT2 7NF, UK
J.Derrick@ukc.ac.uk

Abstract. In this paper we survey recent work on generalising refinement in a state-based setting. Such generalisations challenge a number of assumptions embedded in the standard formalisation of refinement in a language such as Z, and lead to simulation conditions that allow one to verify a refinement in a number of different contexts.

1 Introduction

State-based specification languages have turned out to be a highly effective means to capture abstract requirements. This has motivated theoretical work on foundational issues as well as more practical work and industrial application. Z is a good representative example of a state-based language and it has proved to be one of the most enduring languages, probably because it was born from a combination of simple mathematical elegance together with flexible and useful structuring for the specifier. This has resulted in a specification language which has had diverse application.

It also benefits from a theory of *refinement*. Refinement is the process of development, whereby abstract specifications are turned into more concrete designs. Refinement in Z arises from a simple relational interpretation of specifications, and the foundational work on refinement in a relational setting took place some 15 years ago. This led to the definition of refinement in Z, VDM and other languages.

Recently there has been renewed interest in data refinement, simulations and their realisation in Z. In part this has arisen due to new applications, but also combinations of formal and informal languages have led to work on the relationship between different refinement relations. These new applications have meant that the theory of refinement has had to be adapted to cope with new domains, and this has led to a number of interesting generalisations of refinement. The purpose of this paper is to survey some of them.

2 Example

To motivate the definition of refinement, consider the specification of a watch. We might chose to describe it using the following state space and initialisation

TimeHM

$hrs : 0..23$
$mins : 0..59$

AlarmHM

$alarm : 0..23 \times 0..59$

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 33–56, 2003.
© Springer-Verlag Berlin Heidelberg 2003

InitHM

$$TimeHM'$$
$$AlarmHM'$$
$$hrs' = 0 \land mins' = 0 \land alarm' = (0, 0)$$

The full state is given by $TimeHM \land AlarmHM$, however, it is convenient to separate out the two components $TimeHM$ and $AlarmHM$, and the flexibility of Z allows one to do this.

A collection of operations describe the functionality of the watch. We specify two operations, the first to reset the time: $ResetTime \land \Xi AlarmHM$, and the second to reset the alarm: $ResetAlarm \land \Xi TimeHM$, where $ResetTime$ and $ResetAlarm$ are given by:

ResetTime

$$\Delta TimeHM$$
$$hrs? : 0..23$$
$$mins? : 0..59$$
$$hrs' = hrs?$$
$$mins' = mins?$$

ResetAlarm

$$\Delta AlarmHM$$
$$hrs? : 0..23$$
$$mins? : 0..59$$
$$alarm' = (hrs?, mins?)$$

Operations to display the current time and the current alarm setting are also described. Note that the presence of an operation *ShowTime* models the situation where the time is only displayed when the operation is invoked, we will return to this point later.

ShowTime

$$\Xi TimeHM$$
$$hrs! : 0..23$$
$$mins! : 0..59$$
$$hrs = hrs!$$
$$mins = mins!$$

ShowAlarm

$$\Xi AlarmHM$$
$$alarm! : 0..23 \times 0..59$$
$$alarm! = alarm$$

In order to understand where refinement comes from we first need to link Z specifications to an underlying relational model in which our discussion of refinement will take place. How it works is as follows. We think of a Z specification as defining a number of *programs*, where each program consists of an initialisation followed by a sequence of operations. This will define a relation over some global state space. Thus the specification above will define programs such as the following (this is illustrative - technicalities will alter the formal definition):

$$InitHM \fatsemi ResetTime \fatsemi ShowTime \fatsemi ShowTime \fatsemi ShowAlarm \fatsemi ResetAlarm$$

A check for refinement will then constitute a consistency check of all concrete programs with their abstract counterparts. There are two technicalities to deal with: in order to make the Z specification into a relation we will have to deal with the potential partiality of operations, and also consider the role of input and output. These are discussed below.

3 A Relational Interpretation of a Z Specification

Given a Z specification, we can think of it as a data type, defined as a tuple $(State, Init, \{Op_i\}_{i \in I})$. The operations Op_i are defined in terms of its before-state $State$ and its after-state $State'$. The initialisation is also expressed in terms of an after-state $State'$.

In a specification with no input or output, we can consider a specification as defining a program over a global state which contains no information, i.e., a one point domain, defined as $\mathsf{G} == \{*\}$. The initialisation will be the total relation from the global state $*$ to any value in the set of initial states, i.e.[1]: $\mathsf{Init} == \{Init \bullet * \mapsto \theta State'\}$. For example, the initialisation $InitHM$ defines a relation:

$$\mathsf{Init} == \{* \mapsto (\!| \, hrs = 0, mins = 0, alarm = (0,0) \, |\!)\}$$

Z operations are also interpreted as relations: $\mathsf{Op} == \{Op \bullet \theta State \mapsto \theta State'\}$. We will also add a *finalisation* to the relational counterpart of the Z specification and this finalisation defines what is observable in the specification. In the absence of input and output, finalisations will therefore simply be a relation from the local state to the one point domain G.

This embedding of schemas into relations works for schemas that have no input or output. But how do we turn an operation schema that has input and output into a relation? Well, in such a context, Z operations will be modelled as relations on sequences whose elements are produced and consumed one by one by the operations of the ADT. As these inputs and outputs form part of the *observable* behaviour of the ADT, those sequences will need to be part of the *global* state. They also need to be available for the ADT to change, so they will also be part of the *local* state. Initialisation and finalisation will be concerned with copying these sequences between the global and local state.

We will need the following notation [11]. Denoting $S \vee \neg S$ by ΣS, the *input signature* of an operation schema Op on state $State$ is its projection on input components only, i.e., $?Op = \Sigma(\exists\, State \bullet \text{pre}\ Op)$. Similarly, the *output signature* of an operation schema Op on state $State$ is its projection on output components only, i.e., $!Op = \Sigma(\exists \,\text{pre}\ Op;\ State' \bullet Op)$.

For simplicity, we assume that all operations Op_i have input of type $Input$, and output of type $Output$. The global state now only contains sequences of inputs and outputs. The local state contains both of those plus a representation of the Z ADT state:

$$\mathsf{G} == \text{seq}\ Input \times \text{seq}\ Output$$
$$\mathsf{State} == \text{seq}\ Input \times \text{seq}\ Output \times State$$

The initialisation transfers the sequence of inputs from the global state to the local state, and picks an initial local state that satisfies the specification's

[1] In Z, $\theta State'$ produces the binding which is represented by the components in $State'$, but removing the primes on the labels.

initialisation. The finalisation discards the remaining input sequence and local state, and returns the output sequence produced by the program.

$$\mathsf{Init} == \{Init;\ is : \mathrm{seq}\, Input;\ os : \mathrm{seq}\, Output \bullet (is, os) \mapsto (is, \langle\rangle, \theta State')\}$$
$$\mathsf{Fin} == \{State;\ is : \mathrm{seq}\, Input;\ os : \mathrm{seq}\, Output \bullet (is, os, \theta State) \mapsto (\langle\rangle, os)\}$$

To define the effect of an operation Op on the modified local state the first element is taken from the input sequence, and used as the input ($= \theta?Op$) for the operation. The remainder of the input sequence is left for the following operations. The output produced by the operation ($\theta!Op$) is appended to the output sequence. The state is transformed according to the operation.

$$\mathsf{Op}_i == \{Op_i;\ is : \mathrm{seq}\, Input;\ os : \mathrm{seq}\, Output \bullet$$
$$(\langle \theta?Op_i \rangle \frown is, os, \theta State) \mapsto (is, os \frown \langle \theta!Op_i \rangle, \theta State')\}$$

So, for example, we can think of the operation *ShowAlarm* as defining a relation with elements such as

$$(\langle (\!|\,|\!) \rangle, \langle\,\rangle, (\!|\ hrs = 7, mins = 32, alarm = (8, 54)\ |\!)) \mapsto$$
$$(\langle\,\rangle, \langle (\!|\ alarm! = (8, 54)\ |\!) \rangle, (\!|\ hrs = 7, mins = 32, alarm = (8, 54)\ |\!))$$

The situation we are at is the following. A Z specification defines a number of potential programs, each program being a relation over a global state, which in general consists of the sequences of inputs and outputs used by the program. We now define refinement for a Z specification in terms of these potential programs.

3.1 Refinement

The theory of refinement we will use is defined for *total* relations, however, the above embedding produces some potential partiality. This arises because in languages like Z, operations are often specified by *partial* relations. There are at least two different interpretations of partial operations in the world of state based systems.

In the *contract* or *non-blocking* approach to ADTs, the domain (precondition) of an operation describes the area within which the operation should be guaranteed to deliver a well-defined result, as specified by the relation. Outside that domain, the operation *may* be applied, but may return any value, even an undefined one (modelling, e.g., non-termination). In the *behavioural* or *blocking* approach to ADTs, operations may not be applied outside their precondition; doing so anyway leads to an undefined result. The precondition is often called the *guard* in this context.

Traditionally, the first approach is the more common one in Z, although it depends on the area of application as to which of the two interpretations should be used. (Indeed they are not necessarily exclusive [20].)

In both approaches the simulation rules for partial operations are derived from those for total operations by modelling partial relations on a set S as total relations on a set $\mathsf{S}_\perp$, which is S extended with a distinguished value $\perp$

not in S. The construction is slightly different between the blocking and non-blocking models to code up the different interpretations of the precondition of an operation. Details in [11].

We can therefore think of a Z specification as defining a data type, i.e., a quadruple $(\mathsf{State}, \mathsf{Init}, \{\mathsf{Op}_i\}_{i\in I}, \mathsf{Fin})$, where the operations $\{\mathsf{Op}_i\}$, indexed by $i \in I$, are total relations on the set State; Init is a relation from G to State; Fin is a relation from State to G.

The potential programs of a specification are given by the sequential composition of initialisation, operations and finalisations. For example, for a sequence p over I (i.e., a sequence of operation names), and data type D, p_D denotes the program over D characterised by p. E.g., if $\mathsf{p} = \langle \mathsf{p}_1, ..., \mathsf{p}_n \rangle$ then $\mathsf{p}_\mathsf{D} = \mathsf{Init} \fatsemi \mathsf{Op}_{\mathsf{p}_1} \fatsemi ... \fatsemi \mathsf{Op}_{\mathsf{p}_n} \fatsemi \mathsf{Fin}$. We then use the standard definition of refinement.

Definition 1 (Data refinement) *For data types* A *and* C, C refines A *iff for each finite sequence* p *over* I, $\mathsf{p}_\mathsf{C} \subseteq \mathsf{p}_\mathsf{A}$. □

However, verifying such refinements is non-trivial due to the quantification over all programs. The standard solution is to use simulations [8], where a step-by-step comparison is made of each operation in the data types.

In making this comparison one needs to link the local state spaces, and such a link is known as a simulation, or a retrieve relation, or an abstraction relation, or a coupling invariant. Figure 1 represents the situation where the requirement is that the triangles and squares semi-commute.

There are two key ways to make this diagram semi-commute, known as *downward* and *upward* simulations. These are useful since simulations are sound and jointly complete [17, 8] and thus provide a tractable way to verify refinements.

Definition 2 (Downward simulation)
Let $\mathsf{A} = (\mathsf{AState}, \mathsf{AInit}, \{\mathsf{AOp}_i\}_{i\in I}, \mathsf{AFin})$ *and* $\mathsf{C} = (\mathsf{CState}, \mathsf{CInit}, \{\mathsf{COp}_i\}_{i\in I}, \mathsf{CFin})$ *be data types. A* downward *simulation is a relation* R *from* AState *to* CState

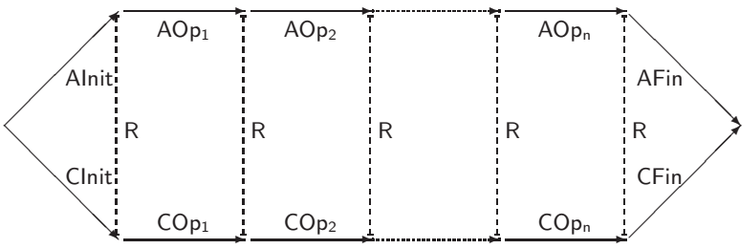


Fig. 1. Data refinement with simulations

satisfying

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$\forall\, i : I \bullet \mathsf{R} \fatsemi \{\mathsf{COp}_i\} \subseteq \{\mathsf{AOp}_i\} \fatsemi \mathsf{R}$$

If such a simulation exists, we also say that C *is a downward simulation of* A. □

There is an analogous definition for upward simulations, however, for the remainder of this paper, we will largely concentrate on downward simulations. The constructions we consider can universally be applied to upwards simulations as well, except where we explicitly mention it.

As we have already made clear, these refinement rules are defined for total relations. Hence, when we have partial relations in a specification we would have to totalise them first before we applied this definition of refinement. However, we can extract the underlying requirement on the partial relations and avoid any need to explicitly refer to $\bot$. Doing this results in the following definition.

Definition 3 (Downward simulation for partial relations)
Let $\mathsf{A} = (\mathsf{AState}, \mathsf{AInit}, \{\mathsf{AOp}_i\}_{i\in I}, \mathsf{AFin})$ *and* $\mathsf{C} = (\mathsf{CState}, \mathsf{CInit}, \{\mathsf{COp}_i\}_{i\in I}, \mathsf{CFin})$ *be data types where the operations may be partial. A* downward *simulation is a relation* R *from* AState *to* CState *satisfying, in the non-blocking interpretation*

$$\mathsf{CInit} \subseteq \mathsf{AInit} \fatsemi \mathsf{R}$$
$$\mathsf{R} \fatsemi \mathsf{CFin} \subseteq \mathsf{AFin}$$
$$ran(dom\, \mathsf{AFin} \lhd \mathsf{R}) \subseteq dom\, \mathsf{CFin}$$
$$\forall\, i : I \bullet ran(dom\, \mathsf{AOp}_i \lhd \mathsf{R}) \subseteq dom\, \mathsf{COp}_i$$
$$\forall\, i : I \bullet (dom\, \mathsf{AOp}_i \lhd \mathsf{R}) \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$$

In the blocking interpretation, correctness (i.e., the final condition) is strengthened to: $\forall\, i : I \bullet \mathsf{R} \fatsemi \mathsf{COp}_i \subseteq \mathsf{AOp}_i \fatsemi \mathsf{R}$. □

3.2 Refinement in Z

We now have a relational interpretation of a Z specification, and a definition of downward simulation for use on partial relations. The next thing to do is to translate the relational refinement conditions into refinement conditions which work at the syntactic level in Z, i.e., exploit the power of the Z schema calculus. This is straightforward, but we note that the conditions on the finalisation are always satisfied in this Z interpretation. We will return to this point later.

Definition 4 (Standard downward simulation in Z)
Let $A = (AState, AInit, \{AOp_i\}_{i\in I})$ *and* $C = (CState, CInit, \{COp_i\}_{i\in I})$ *be Z data types. The relation* R *on* $AState \wedge CState$ *is a* downward simulation *from* A *to* C *if for all* $i \in I$*:*

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet AInit \wedge R'$$
$$\forall\, AState;\ CState \bullet pre\, AOp_i \wedge R \Rightarrow pre\, COp_i$$
$$\forall\, AState;\ CState;\ CState' \bullet pre\, AOp_i \wedge R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

The above rules assume the non-blocking interpretation of ADTs. In the blocking interpretation, the rule for correctness becomes

$$\forall\, AState;\ CState;\ CState' \bullet R \wedge COp_i \Rightarrow \exists\, AState' \bullet R' \wedge AOp_i$$

When this blocking correctness condition holds, the applicability condition above is equivalent to $\forall\, AState;\ CState \bullet R \Rightarrow (pre\, AOp_i \Leftrightarrow pre\, COp_i)$. □

As an example, we could refine the watch specification to one with the following state space: $CTimeHM == [time : \mathbb{N} \times \mathbb{N}]$, and initialisation and operations:

CInitHM

$CTimeHM'$
$AlarmHM'$

$time' = (0, 0) \wedge alarm' = (0, 0)$

ResetTime

$\Delta CTimeHM$
$hrs? : 0..23$
$mins? : 0..59$

$time' = (hrs?, mins?)$

and so on. This is a simple refinement of the original specification using the retrieve relation $R == [TimeHM;\ CTimeHM \mid time = (hrs, mins)]$.

This gives the basic definition of refinement, as described for example in [25, 11], and examples of refinements of this sort are well documented in the literature. There has, however, been recent work which has generalised the above framework in a number of ways. These all have the aim of supporting a number of types of refinement which hitherto we have not been able to support.

There are two strands to the generalisations we make in the sections 4 to 7. The first allows more general relationships between the abstract and concrete programs, often also relaxing the idea that data types must be conformal (i.e., they use the same index set for the operations in both the abstract and concrete specification). Non-atomic refinement and weak refinement fall into this category. The second strand changes the notion of observation that is made of a data type, and IO and grey box refinement fall into this category. A fuller treatment of generalisations discussed is contained in [11] and an overview of them in [4].

4 IO Refinement

In the theory of refinement presented so far, any operations in the abstract and concrete specifications are required to have identical inputs and outputs. However, it is clear that in practice refinement does not always posses this property, e.g., an input selected from an enumerated type might ultimately be implemented by a radio button. A refinement should be able to cope with such changes of representation and document them precisely.

IO refinement does this and, in fact, does so by generalising a rather arbitrary assumption in the original relational formulation of refinement. Specifically, refinement was defined via a simulation between the two local state spaces, where the state space of a specification is augmented with input and output sequences. The simulation (which becomes the retrieve relation when written in Z) consisted

of a simulation on the state space, plus identity relations between the input and output sequences. Similarly, the initialisation and finalisation simply copy the relevant sequence between the global and local state.

The restriction to identities between input and output sequences is artificial, and we can thus generalise them to simulations between the input and output sequences. Thus our now generalised simulation relates the abstract and concrete state as before, but now instead of just copying input and output sequences it becomes a more general relation between them. But not every relation would be suitable here. For example, in order to rule out any undesirable cross-link between the hidden local state and the observable global state (to which input and output belong in some sense) we exclude any relation that would use the (concrete or abstract) state to influence the transformation of input and output.

We also ask that the input/output for a concrete operation is fully determined by the input/output for its abstract counterpart, not by any other inputs or outputs (which would amount to looking in the future or in the past). This is achieved by only allowing transformations to be defined in terms of total element-wise transformations. Finally, we require that the IO transformations are total, and that they do not reduce the length of the input sequence, as this might leave the concrete program in a state where there is no input left to consume.

Then, using these ideas we can derive simulation rules for IO refinement between two Z specifications. There are two technicalities to deal with. The first technicality is how to represent the transformations of input and output, and we do this by using a notion of IO-transformers.

Definition 5 (IO-Transformers)
An input transformer *IT for an operation Op is a schema whose outputs exactly match Op's inputs, i.e. for every name x? in Op there is a name x! in IT and vice versa, and all of whose other names are inputs.* Output transformers *are defined analogously.* □

For a formal definition of these notions, and how IO transformers are applied using the schema piping operator $\gg$, cf. [3]. These transformations are formally applied to input and output sequences in a pointwise fashion and we can use this formalisation to derive IO refinement in the Z schema calculus.

The second technicality, which we won't formalise, concerns the need to preserve the transformations made to input and output, since unlike the states linked by the normal retrieve relation, these are in some sense observable. Details of this are given in Chapter 10 of [11]. Eliding these details, we have the following definition.

Definition 6 (IO downward simulation)
Let $A = (AState, AInit, \{AOp_i\}_{i \in I})$ *and* $C = (CState, CInit, \{COp_i\}_{i \in I})$ *be conformal ADTs. Let for* $i \in I$, IT_i *be an input transformer for* COp_i *which is total on* $?AOp_i$. *Let* OT_i *be a total injective output transformer for* AOp_i. *The retrieve relation R defines a downward simulation between A and C if for all*

$i \in I$:

$$\forall\, CState' \bullet CInit \Rightarrow \exists\, AState' \bullet R' \land AInit$$
$$\forall\, CState;\ AState;\ ?COp_i \bullet pre(\overline{IT_i} \gg AOp_i) \land R \Rightarrow pre\, COp_i$$
$$\forall\, AState;\ ?AOp_i;\ CState;\ CState';\ !COp_i \bullet$$
$$R \land pre\, AOp_i \land (IT_i \gg COp_i) \Rightarrow \exists\, AState' \bullet R' \land (AOp_i \gg OT_i) \qquad \square$$

For example, we can use IO refinement to refine the watch specification to one with an analogue display. To do so let $ANGLE == 0..359$, which represents the angle between a hand and the positive y-axis. We define a function to convert between the two displays

$$Angle : 0..23 \times 0..59 \to ANGLE \times ANGLE$$
$$Angle(x, y).1 = 30 * (x \bmod 12) + y \operatorname{div} 2$$
$$Angle(x, y).2 = 6 * y$$

To model the analogue watch, we now replace the original *ShowTime* by

ShowTimeHands
$$\Xi TimeHM$$
$$hrs! : ANGLE$$
$$mins! : ANGLE$$
$$(hrs!, mins!) = Angle(hrs, mins)$$

This refinement changes the type of outputs, and indeed is a valid IO refinement. We can prove this using a theorem which allows one to construct valid IO refinements [3, 11]. Formally it says that an operation AOp is IO refined by $COp = \overline{IT} \gg AOp \gg OT$ when the following conditions hold:

- OT is a total injective output transformer for AOp;
- IT is an injective input transformer for COp.

We apply this to $ShowTimeHands = ShowTime \gg OT$, where OT essentially applies the function *Angle* to its inputs, and outputs the result. OT is total (defined at every "time" instant) and injective (every hour and minute has a unique representation in terms of the hands). Observe that the representation using just the big hand is insufficient: the function div 2 is not injective.

Similarly, injectivity of *Angle* is also sufficient to prove that an analogous equivalent of *ResetTime* is a valid IO refinement:

ResetTimeHands
$$\Delta TimeHM$$
$$hrs? : ANGLE$$
$$mins? : ANGLE$$
$$(hrs?, mins?) \in \operatorname{ran} Angle$$
$$(hrs?, mins?) = Angle(hrs', mins')$$

The restriction on $(hrs?, mins?)$ reflects the fact that some combinations of angles are impossible. However, this also includes angles that are possible on real analogue watches, but which would not reflect an exact number of minutes on the digital watch, e.g., $(0, 3)$ might be said to represent half a minute past noon. We might want to assume the blocking interpretation of this operation in order to prevent $(0, 3)$ getting linked to an arbitrary time in refinement.

Other simple theorems about IO refinement can also be derived. For example, for any ADT, adding a declaration of a new input or a new output (from a non-empty set) to one of the operations constitutes a valid IO downward simulation, and these are often useful in similar situations and constructions.

5 Grey Box Refinement

The specification of the *ShowTime* operation given above was not entirely satisfactory. Unless we were really describing an LED display we would like to consider the display of the time to be continuous, however *ShowTime* only displays the time when the operation is invoked. In addition, the *ResetTime* operation states only that the components *hrs* and *mins* may be modified by replacing them with inputs with the same base names, and this is a very common template.

Grey box specification [2] provides a way of modelling these types of operations by allowing state components to be observable or modifiable. The assumption that we will alter in this specification style is the representation hiding implied by the abstract or black box nature of the state. Thus we will consider data types which are not entirely abstract, but which in fact may be partially concrete in some sense.

Thus a grey box data type is a variant on the standard specification, for which some of the components of the state schema are declared to be observable or modifiable. As a consequence, the state of a grey box data type is partitioned into three parts, all of them obtained from the complete state through schema projection. The private components are those which are not observable. They are specified only implicitly, *viz.* as those components not in the other two subsets. Both the readable components and the modifiable components can be observed, the modifiable components can be changed as well.

Definition 7 (Grey box)
A grey box data type *is a tuple* $(State, Read, RW, Init, Ops)$ *such that* $(State, Init, Ops)$ *is a black box data type (called the* underlying *ADT), and Read and RW are disjoint projections of State, denoting the read-only components and the modifiable (read-write) components.* □

For example, a grey box specification of the watch is $(TimeHM \wedge AlarmHM, AlarmHM, TimeHM, InitHM, \{ResetAlarm, AddMin, Alarm\})$. The second component of the grey box denotes the schema of all observable, but not directly modifiable, components. Due to the type of *alarm* not matching that of the inputs of *ResetAlarm*, the latter operation still needs to be explicitly

included and *AlarmHM* is considered observable but not modifiable. The third component of the grey box represents the modifiability and observability of *TimeHM*, replacing operations *ResetTime* and *ShowTime*. We include state components in the second and third components of the grey box data type so that operations to observe and modify are made implicit.

Grey box specifications allow only a limited range of refinements, due to the observable state components having to remain part of the state in further refinement. In other words, they are good for implementation specifications, but less good for abstract specifications. The same disadvantage does not hold for *display box* specifications which are ADTs some of whose "operations" (*displays*) do not need to mention the after-state because it is implicitly assumed to be equal to the before-state.

Displays will be written as schemas which do not refer to the after-state $State'$, and which have no inputs. Their interpretation is to add $\Xi State$ to displays.

Definition 8 (Display box)
A display box data type *is a tuple* $(State, Ds, Init, Ops)$ *such that* $(State, Init, Ops)$ *is a black box data type (the* underlying black box*), and every element* D *of the set* Ds *is a schema on* $State$ *and some other ("output") type, such that* D *is total, i.e.,* $D \upharpoonright State = State$. □

In our watch example two observation operations could be defined as displays, for example, *ShowTime* would be

$$
\begin{array}{|l}
\hline
DispTime_A \\
TimeHM \land AlarmHM \\
hrs! : 0 \mathinner{..} 23 \\
mins! : 0 \mathinner{..} 59 \\
\hline
hrs = hrs! \\
mins = mins! \\
\hline
\end{array}
$$

We could now replace the standard clock by an unbounded clock, as in: $Eternal == [mins_0 : \mathbb{N}]$. Then the weakest refinement of $DispTime_A$ using a retrieve relation R can be calculated using standard techniques, producing $DispTime_C$:

$$
\begin{array}{|l}
\hline
R \\
Eternal \\
TimeHM \\
\hline
hrs = (mins_0 \operatorname{div} 60) \bmod 24 \\
mins = mins_0 \bmod 60 \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\hline
DispTime_C \\
Eternal \land AlarmHM \\
hrs! : 0 \mathinner{..} 23 \\
mins! : 0 \mathinner{..} 59 \\
\hline
hrs! = (mins_0 \operatorname{div} 60) \bmod 24 \\
mins! = mins_0 \bmod 60 \\
\hline
\end{array}
$$

As this operation is total, it is a display. Moreover, it is deterministic, so any display box refinement using R as the retrieve relation will refine display $DispTime_A$ to $DispTime_C$.

6 Weak Refinement

We now turn our attention from observability to conformity, and look at generalisations that change the number of operations in the abstract and concrete specifications. The first way to do this is to look at the consequences of the presence of *internal operations* in either the abstract or concrete specifications.

As an example, consider augmenting the watch specification with a component that will count in seconds. To do so we use the new state space

$$\begin{array}{|l}\hline TimeHMS \\ TimeHM \\ secs : 0..59 \\ \hline\end{array} \qquad \begin{array}{|l}\hline InitHMS \\ TimeHMS' \\ InitHM \\ \hline secs' = 0 \\ \hline\end{array}$$

With this state space we will only change the *AddMin* operation and add an internal *Tick* operation to increment the seconds. All other operations remain the same. The two operations of interest are described by

$$\begin{array}{|l}\hline Tick \\ \Delta TimeHMS \\ \Xi TimeHM \\ \hline secs' = secs + 1 \\ secs \neq 59 \\ \hline\end{array} \qquad \begin{array}{|l}\hline AddMinS \\ \Delta TimeHMS \\ \hline secs = 59 \land secs' = 0 \\ mins' = (mins + 1) \bmod 60 \\ \textbf{let}\ \alpha == (mins + 1) \operatorname{div} 60 \bullet \\ \quad hrs' = hrs + \alpha \\ \hline\end{array}$$

The definition of refinement in the presence of such internal operations has deep resonance with the treatment of internal events in a process algebraic framework. This has been investigated in some depth in [10].

In terms of defining refinement for Z specifications containing internal operations we first note that one could imagine two approaches to refinement of internal operations: either use the existing refinement rules (treating the internal operations separately), or generalise the existing rules.

The first of these involves using stuttering steps in the verification of a refinement. In the context of a Z specification a stuttering step is an operation which does not change the abstract state, i.e., *skip* will be $\Xi State$ for a state space *State*. The requirement on internal operations will be that they refine *skip*. Since a stuttering step is total the requirements for refining to a concrete internal operation *CIOp* are $R \land CIOp \Rightarrow skip_A$ and $R \Rightarrow \mathrm{pre}\ CIOp$, where $skip_A$ is the abstract stuttering step.

However, there is a problem with this approach due to the interpretation of internal operations and the potential for divergence through their introduction. Internal operations are interpreted as being internal because they are not under the control of the environment of the ADT, and in some way they represent operations that the system can invoke whenever their preconditions hold. Because of this we do not want divergence to be introduced by a refinement as a result of

their infinitely repeated application. However, refinements of *skip* have precisely this property since *skip* is total.

Related to this issue is a question concerning the meaning of a precondition of an internal operation. Because we wish internal operations to be applicable only at certain values in the state space, we need to take an interpretation which says they are *not* applicable outside their precondition. For example, the precondition of *Tick* is $sec \neq 59$, and incorrect refinements would result if we could weaken its precondition and perform arbitrary behaviour when $sec = 59$ as would be allowed under the standard interpretation of refinement. Therefore internal operations may not be applied outside their precondition.

The solution to these issues is to use the approach used in process algebraic languages. Thus refinement is viewed in terms of the external behaviour of the specification, and we define weak refinement rules [12, 13] that ensure the observable behaviour of the concrete specification is a valid refinement of the observable behaviour of the abstract specification. In particular we move from the application of a single observable operation *Op* to a situation where a finite number of internal operations are allowed before and after the occurrence of that operation.

Thus we assume that the sets of observable operations in a refinement are conformal, but we now extend the notion of a data type to include a set of internal operations $\{IOp_j\}_{j \in J}$ for some index set J. The definition of conformal is adjusted to require that the indexing sets of observable operations coincide. However, we make no requirements on the conformity of the internal operations, and indeed weak refinement will allow the introduction or removal of arbitrary number of internal operations during a refinement.

We can re-formulate the basic condition of data refinement to take internal operations into account, and ask that for any given program (i.e., choice of observable operations), every concrete program (possibly involving internal operations) is contained in some abstract (possibly involving internal operations but using the same choice of observable operations).

We can then, as usual, adapt the simulation rules to verify refinements. In order to do so we encode the idea of a finite number of internal operations before and after an observable operation directly in the Z schema calculus. In order to avoid quantifications over sequences of internal operations in the definition of weak refinement, we encode "all possible finite internal evolution" for a specification as a *single* operation *Int* (allowing us to replace operations *Op* by $Int \fatsemi Op \fatsemi Int$ to allow for internal evolution before and after). (Gory details of how to do this are in [13, 11].)

The definition of a weak downward simulation then has the same form as the standard Z definition. Internal operations in the concrete and abstract specifications are differentiated by using the subscripts C and A on *Int*. To prevent the introduction of divergence we use two additional conditions D1 and D2.

Definition 9 (Weak downward simulation)
Let $A = (AState, AInit, \{AOp_i\}_{i \in I \cup J})$ *and* $C = (CState, CInit, \{COp_i\}_{i \in I \cup K})$ *be Z data types, where* J *and* K *(the index sets denoting internal operations)*

are both disjoint from I. *Then the relation* R *on* $AState \wedge CState$ *is a* weak downward simulation *from* A *to* C *if* $\forall i : I$

$$\forall CState' \bullet (CInit \fatsemi Int_C) \Rightarrow \exists AState' \bullet (AInit \fatsemi Int_A) \wedge R'$$
$$\forall AState;\ CState \bullet pre(Int_A \fatsemi AOp_i) \wedge R \Rightarrow pre(Int_C \fatsemi COp_i)$$
$$\forall AState;\ CState;\ CState' \bullet pre(Int_A \fatsemi AOp_i) \wedge R \wedge (Int_C \fatsemi COp_i \fatsemi Int_C) \Rightarrow \exists AState' \bullet R' \wedge (Int_A \fatsemi AOp_i \fatsemi Int_A)$$

In addition, we require the existence of a well-founded set WF *with partial order* $<$, *and a variant* E *which is an expression in the concrete state variables satisfying the following conditions:*

D1 $R \vdash E \in WF$
D2 $\forall i : K \bullet R \wedge COp_i \vdash E' < E$ □

Note that although internal operations decrease the variant, there are no constraints on observable operations, which are allowed to increase the variant. This means that an internal operation can be invoked an infinite number of times, but not in an infinite sequence between observable operations.

Then using this definition we can show that the new watch specification (i.e., the one with *Tick* in it) is a weak refinement of the original watch description. To verify it formally we would use a retrieve relation that is the identity on *TimeHM*, the conditions are then easily discharged.

For example, divergence cannot occur since *Tick* can clearly only be applied a finite (e.g., less than 60) number of times between observable operations. In addition, the *AddMin* operation is effectively refined by *AddMinS* and a number of internal *Tick* operations. For example, whenever the precondition of *AddMin* holds then either the precondition of *AddMinS* holds or *Tick*s will lead to a state where the precondition holds.

Further examples of specifications containing internal operations and their refinement are given in [13, 11].

7 Non-atomic Refinement

The previous section showed how we could refine specifications containing internal operations. In this section we look at a related issue and consider how to liberalise the definition with the aim of supporting non-conformal refinements. Such refinements, known as non-atomic refinements, have received considerable attention in the process algebraic world under the name of action refinement [7, 1]. Here we consider related work in the state-based domain.

The generalisation of refinement and simulations we consider here for state-based languages is given a fuller treatment in [9, 14]. The basic approach is that instead of refining an abstract operation into a *single* concrete operation, it is now potentially refined to a *sequence* of concrete operations. To support this we need a modified definition of refinement because the programs in the abstract and concrete specifications will now have a slightly different form. And this will give rise to a new set of simulation conditions.

In fact, for the full generality of the definition we will need the technique of IO refinement to solve some of the technical problems. The issue occurs if the abstract operation being decomposed has inputs or outputs. Then these inputs and outputs may be spread over the sequence of concrete operations, and therefore IO-transformers are needed to verify that the simulation conditions hold.

As an example of the type of decomposition we wish to support, consider the *ResetTime* and *ResetAlarm* operations in the watch specification. These descriptions are, in fact, unrealistic since usually digital watches do not have an interface that allows you to enter a new arbitrary time to which it is atomically set. Instead there is a process of pressing a selection of buttons repeatedly which increments the hours and minutes until the correct time is reached. However, it would be wrong to describe that process at the initial level of abstraction not least because different watches will want to use different mechanisms (e.g., resetting an analogue watch is completely different from resetting a digital one), and moreover, such a description does not capture the essential process in a clear way.

To illustrate this we refine the initial description to one with a simple display which has two buttons *ButA* and *ButB* which will be used to implement the functionality. The original collection of operations *ShowTime, ShowAlarm, ResetTime, ResetAlarm, AddMin, Alarm* will thus be refined to *Display, ButA, ButB, AddMin, Alarm*. The new specification will use a number of modes, given by:

$$MODE ::= display_t \mid display_a \mid change_h \mid change_m \mid change_a$$

The state space *AlarmHM* is unchanged, and we add a single *mode* to *TimeHM*:

TimeDG

$TimeHM$
$mode : MODE$

InitDG

$TimeDG'$
$InitHM$

AddMin is left unchanged apart from requiring that it does not change the *mode* of the watch. The watch has a single display which can either show the time or the alarm (but not both) at any particular moment. That is, *ShowTime* and *ShowAlarm* are refined to a single operation *Display*.

Display

$\Xi TimeDG$
$\Xi AlarmHM$
$display! : 0..23 \times 0..59$

$(mode \in \{display_t, change_h, change_m\} \land display! = (hrs, mins)) \lor$
$(mode \in \{display_a, change_a\} \land display! = alarm)$

We use *ButA* to switch between the modes, thus allowing different functionality to be achieved with the second button.

ButA

$\Delta TimeDG$
$\Xi TimeHM$
$\Xi AlarmHM$

$mode = display_t \Rightarrow mode' = display_a$
$mode = display_a \Rightarrow mode' = change_h$
$mode = change_h \Rightarrow mode' = change_m$
$mode = change_m \Rightarrow mode' = change_a$
$mode = change_a \Rightarrow mode' = display_t$

ButB is more complex: if it is pressed when in either of the display modes, nothing happens. However, if the watch is in $change_h$ mode, then it will increment the hours by one, and similarly for $change_m$ mode. Finally to reset the alarm we have to move through the minutes and hours by pressing *ButB* in $change_a$ mode: $ButB == ButBDisp \lor ButBHrs \lor ButBMins \lor ButBAlm$.

ButBDisp

$\Xi TimeDG$
$\Xi AlarmHM$

$mode \in \{display_t, display_a\}$

ButBHrs

$\Delta TimeDG$
$\Xi AlarmHM$

$mode = change_h \land mode' = mode$
$mins' = mins$
$hrs' = (hrs + 1) \bmod 24$

ButBMins

$\Delta TimeDG$
$\Xi AlarmHM$

$mode = change_m$
$mode' = mode$
$hrs' = hrs$
$mins' = (mins + 1) \bmod 60$

ButBAlm

$\Xi TimeDG$
$\Delta AlarmHM$

$mode = change_a \land mode' = mode$
let $\beta == (mins + 1) \bmod 60;$
$\gamma == (mins + 1) \operatorname{div} 60;$
$\alpha == (hrs + \gamma) \bmod 24 \bullet$
$alarm' = (\alpha, \beta)$

This new specification is a non-atomic refinement of the original. For example, *ShowTime* is refined by *Display*, and *ShowAlarm* by $ButA \fatsemi Display \fatsemi ButA^4$ (where Op^n means n compositions of Op).

However, the refinements of *ResetAlarm* and *ResetTime* are more complicated, and, in fact, under the refinement all the inputs in these two operations disappear completely. Here the inputs representing the new time settings have been implemented as a series of pushes to button *ButB* - the number of pushes needed equals the necessary increments in order to reach the new time from the current time. Therefore *ResetTime* is refined by the composition

$$ButA^2 \fatsemi ButB^\alpha \fatsemi ButA \fatsemi ButB^\beta \fatsemi ButA^2$$

Here α and β are the difference between the current and desired hours and minutes respectively, i.e., $\alpha = (hrs? - hrs) \bmod 24$ and $\beta = (mins? - mins) \bmod 60$.

Note that the crucial point about this is that the length of the concrete decomposition needed varies according to the current time and to the new time to which we wish to reset our watch.

It is immediately clear that we will need IO transformers in this refinement since, in the case of *ResetAlarm* or *ResetTime* all of the inputs have completely disappeared in the concrete operations. Before considering in general how this is supported we consider non-atomic refinement without IO transformers.

We are now in a situation where data types A and C have different indexing sets (I_A and I_C respectively) for their observable operations, so to discuss refinement we need to document the relationship between them. For example, suppose this is given as a mapping $\rho : I_A \rightarrow \operatorname{seq} I_C$, so that ρ describes the concrete counterparts for each operation in A. (e.g., if $\rho(7) = \langle 6, 3 \rangle$ then this means that operation AOp_7 is refined by the sequence $COp_6 \fatsemi COp_3$.)

Definition 10 (Non-atomic refinement)
For data types A *and* C, C *non-atomically refines* A *with respect to* ρ *iff for each finite sequence* p *over* I_A, $(\frown/(\rho \circ \mathsf{p}))_{\mathsf{C}} \subseteq \mathsf{p}_{\mathsf{A}}$. □

For example, if AOp_1 is refined by the sequence $COp_1 \fatsemi COp_2$ then action refinement requires that, amongst others, $CInit \fatsemi COp_1 \fatsemi COp_2 \fatsemi CFin \subseteq AInit \fatsemi AOp_1 \fatsemi AFin$.

Simulations can be used to make step-by-step comparisons as before. For the sake of readability suppose that in the two data types the indexes coincide except that the abstract operation AOp is refined by the sequence $COp_1 \fatsemi COp_2$.

Definition 11 (Non-atomic downward simulation without IO trans.)
Let R *be a retrieve relation between data types* $(AState, AInit, \{AOp\})$ *and* $(CState, CInit, \{COp_1, COp_2\})$. *Then* R *is a non-atomic downward simulation if, in addition to the initialisation condition, the following hold.*

$$\forall\, AState;\ CState;\ CState' \bullet pre\, AOp \wedge (COp_1 \fatsemi COp_2) \wedge R \Rightarrow \exists\, AState' \bullet R' \wedge AOp$$
$$\forall\, AState;\ CState \bullet pre\, AOp \wedge R \Rightarrow pre\, COp_1$$
$$\forall\, AState;\ CState;\ CState' \bullet pre\, AOp \wedge R \wedge COp_1 \Rightarrow (pre\, COp_2)'$$ □

The conditions in this definition express the following requirements. The first says that the effect of $COp_1 \fatsemi COp_2$ is consistent with that of AOp (but can of course reduce any non-determinism in AOp). The second says that COp_1 can be invoked whenever AOp can, and the third says that when COp_1 has been completed COp_2 can be invoked (we write $(\text{pre}\, COp_2)'$ so the before-state matches the after-state of COp_1). Informally these are clearly the correct conditions for a refinement of AOp into $COp_1 \fatsemi COp_2$.

The more general case, when we alter the input and output, is dealt with as follows. We will use input and output transformers whose structure is identical

to that of the mapping $\rho : I_A \rightarrow \operatorname{seq} I_C$. This means that there will not necessarily be a 1-1 mapping between the number of abstract inputs/outputs and the number of concrete inputs/outputs. Moreover, in general the input transformations might not be between two operations, but a whole sequence of concrete operations, the length of which is only determined at the time of applicability by the abstract inputs.

Hence we use mappings from an abstract input to a sequence of concrete inputs representing the inputs needed in the decomposition. For example, the following definition expresses the refinement of AOp into a fixed sequence $COp_1 \fatsemi COp_2$.

Definition 12 (Non-atomic downward simulation with IO trans.) *Let R be a retrieve relation between data types $(AState, AInit, \{AOp\})$ and $(CState, CInit, \{COp_1, COp_2\})$. Let IT be an input transformer for $COp_1 \fatsemi COp_2$ which is total on the abstract inputs. Let OT be a total injective output transformer for AOp. Then R defines a non-atomic IO downward simulation if, in addition to the initialisation:*

$$\forall AState;\ CState;\ CState' \bullet pre\, AOp \wedge R\ \wedge (IT \gg COp_1 \fatsemi COp_2) \Rightarrow \exists AState' \bullet R' \wedge (AOp \gg OT)$$
$$\forall AState;\ CState \bullet pre(\overline{IT} \gg AOp) \wedge R \Rightarrow pre\, COp_1$$
$$\forall AState;\ CState \bullet pre(\overline{IT} \gg AOp) \wedge R \wedge COp_1 \Rightarrow (pre\, COp_2)' \qquad \square$$

The IO transformations needed for the refinement of *ShowTime* into *Display*, and *ShowAlarm* into $ButA \fatsemi Display \fatsemi ButA^4$ are relatively simple, but are still not the identity. *ButA* does not have any input/output, so the output of *ShowTime* is represented completely in *Display*. Here, in fact, we need only a simple IO refinement because the two outputs *hrs*! and *mins*! have become a single *display*!, and, because of this, standard refinement would not be sufficient here. Thus we need to use an output transformer OT:

OT

$$hrs? : 0..23$$
$$mins? : 0..59$$
$$display! : 0..23 \times 0..59$$
$$display! = (hrs?, mins?)$$

With this output transformer we can show the conditions in Definition 12 are met. (To do so the retrieve relation we use is the identity on $TimeHM \wedge AlarmHM$ and the *mode* set to $display_t$.) For example, to show that *ShowAlarm* is refined by $ButA \fatsemi Display \fatsemi ButA^4$ we first note that the preconditions involved in this are all true. Thus the conditions reduce to verifying the correctness condition which compares the effect of $ButA \fatsemi Display \fatsemi ButA^4$ with *ShowAlarm*. That is, one has to check:

$$\forall TimeHM \wedge AlarmHM;\ TimeDG \wedge AlarmHM;\ TimeDG' \wedge AlarmHM' \bullet R \wedge ButA \fatsemi Display \fatsemi ButA^4 \Rightarrow \exists TimeHM' \wedge AlarmHM' \bullet R' \wedge (ShowAlarm \gg OT)$$

The input and output transformers needed for the refinement of *ResetAlarm* and *ResetTime* are more complicated since the refinement depends not only on the input value, but also on the current state. The definition of the IO-transformers is thus rather complex, and we do not give details of their construction here.

8 Using Finalisations

The generalisations we have looked at so far have been achieved by relaxing conformity or by generalising the retrieve relation between the two specifications. There is another type of generalisation we can make, one which goes to the heart of the notion of observation, and specifically it looks at the role of finalisations in a specification.

Finalisations, as we have said, represent what is observable in a specification. In our initial relational formalisation they were concerned with making visible the outputs of a program. This is a fairly weak notion of observation, and we can generalise this by considering programs that make stronger observations.

8.1 IO Refinement via Finalisations

One use of generalised finalisations is to derive a form of IO refinement. This is due to Stepney et al as described in [23].

In [23] simulation conditions for input/output refinement are derived by embedding the transformation of output directly into the finalisation (as opposed to using additional retrieve relations as we did above). Thus an explicit finalisation is written in the Z specification directly, and it documents the relationship between the abstract and concrete outputs. In the watch example above, to change between *hrs*!, *mins*! and a single *display*! (as we did in the non-atomic refinement) we could have thus used the finalisation:

Fin

$$hrs! : 0..23$$
$$mins! : 0..59$$
$$display! : 0..23 \times 0..59$$

$$display! = (hrs!, mins!)$$

along with an analogous output retrieve relation. [23] then shows how simulation rules can be derived, where the finalisation condition is now explicitly represented in the schema calculus formulation since we are using non-trivial finalisations.

8.2 Failures-Divergences Refinement via Finalisations

One obvious question to ask of the definition of data refinement that we have used throughout this paper is what is its relationship to refinement in other paradigms. In particular, what is its relationship to process algebraic refinement.

This is interesting since process algebras have many refinement relations defined via differing semantic models (e.g., trace refinement, failures refinement, failures-divergences refinement [21]), whereas a state-based language like Z only has one notion of refinement. Therefore an important research topic is to try to reconcile the differences between a relational notion of refinement and a process algebraic one.

Work relating concurrent and state-based refinement includes work on semantic aspects due to Josephs [19], He [18] and Woodcock and Morgan [26], as well as more recent work by Bolton and Davies [6, 5]. In particular, Bolton and Davies in [5] showed that relational data refinement as verified by the simulation conditions given above does not correspond to failures-divergences refinement, but to a slightly different semantic model called singleton failures. This led to work by the authors which used finalisations to strengthen relational data refinement so that it *does* correspond to failures-divergences refinement. We review briefly here how this is done, full details are given in [15].

Remember that in a failures-divergences semantics (which is the standard semantics of CSP [21]) a process is modelled by its *failures* and its *divergences*. The failures of a process are pairs (t, X) where t is a finite sequence of events that the process may undergo and X is a set of events the process may refuse to perform after undergoing t. The divergences of a process are the sequences of events after which the process may undergo an infinite sequence of internal events, i.e., livelock. Divergences also result from unguarded recursion.

One immediate observation to make is that relational refinement is concerned more with the relation between global input and output, but that this contrasts with a failures-divergences viewpoint in which the event names have an importance not immediately visible in the relational setting. This difference is usually resolved by describing a corresponding process for a relational data type which makes the events observable.

But the primary difference between the two models is the weakness of observations in the relational model, and this is the reason for the difference between data refinement and failures-divergences refinement. Thus in order to strengthen data refinement one needs to make more observations, and the way we do this is to increase the expressiveness of the finalisation, since this represents what is observable.

Thus we will move from finalisations just making the output sequence visible to a situation where we can observe arbitrary information. This will be achieved by adding a component, E, to the finalisation. What goes in E depends on what we want to observe, for example, here we will illustrate the process by letting E represent what operations could be *refused* at the state in which we have just finalised. This will give us failures-divergences refinement, other choices allow one to derive other concurrent refinement relations in a similar fashion. Thus our finalisation will become

$$\{State;\ is : \mathrm{seq}\, Input;\ os : \mathrm{seq}\, Output \bullet (is, os, \theta State) \mapsto (\langle\rangle, os, E)\}$$

For example, to code refusals in the relational setting (in the absence of input and output) we take E to be any subset of the following (where I is the set of

operation names). Our relational interpretation will thus be:

$$\mathsf{Fin} == \{State;\ E : \mathbb{P}\, I \mid (\forall\, i \in E \bullet \neg \,\mathrm{pre}\, Op_i) \bullet \theta State \mapsto E\}$$

Now the crucial aspect of this is that the refinement obligations on finalisations now do *not* disappear when we translate them into the Z schema calculus. Instead they lead to a strengthening of the conditions for *upwards simulations*, the details of this are in [15], but essentially the new condition arising due to this finalisation is (I is the set of operation indices):

$$\forall\, CState \bullet \exists\, AState \bullet \forall\, i : I \bullet\ T \wedge (\mathrm{pre}\, AOp_i \Rightarrow \mathrm{pre}\, COp_i)$$

which is strictly stronger than, and thus subsumes, the standard upwards simulation applicability condition[2]:

$$\forall\, i : I \bullet \forall\, CState \bullet \exists\, AState \bullet\ T \wedge (\mathrm{pre}\, AOp_i \Rightarrow \mathrm{pre}\, COp_i)$$

This can be summarised as follows. The standard upward simulation applicability condition requires that we have to consider pairs of abstract and concrete states for each operation. The finalisation condition, on the other hand, requires that for every abstract state we can find a *single* concrete state such that all the preconditions of the abstract operations imply the preconditions of their concrete counterparts. This is a stronger condition and one that highlights the difference between data refinement and failures-divergences refinement. [15] contains a proof that using this finalisation, data refinement corresponds to failures-divergences refinement.

The process of generalising the finalisation used does not alter the definition of data refinement, rather it changes the embedding of Z into its relational interpretation. Therefore, the form of the simulation rules in a relational setting is unchanged, it is just the expression of the particular finalisation used that makes a difference. Thus since the simulation rules are known to be sound and jointly complete the new formalisation represents a tractable method for verifying failures-divergences refinement.

As commented above, the expression of the simulation rules in the Z schema calculus is more complex in the presence of input and output in the operations. However, the results are the same, that is, data refinement corresponds to failures-divergences refinement and simulation rules can be derived that allow an expression of these conditions within the Z schema calculus.

The presence of outputs does, however, cause complications in their expression. In fact, one has a choice of how to deal with outputs depending on how strongly the environment is coupled to the system under consideration. Specifically, what happens if an operation is applied in a state and input inside its precondition, can it ever be refused? The answer depends on how the environment reacts with it with respect to synchronisation on outputs. There are two (standard) choices:

[2] These are for use in operations without input or output, the full rules are more complex.

- the environment cannot influence the output, and there are refusals due to a particular output being chosen, or
- the environment can influence the output, and there are no such refusals.

The first represents a situation when the outputs of an operation cannot be constrained by the environment: one can refuse all but one of the possible assignment of values to the outputs corresponding to a particular operation and assignment of values to the inputs. The second represents a situation when the coupling between a system and its environment is tighter and the environment can affect non-determinism in the outputs of operations by choosing a value to synchronise on if one can be found. The former is called a *demonic* choice of outputs, and the latter an *angelic* choice [22], and the refusals one gets differ slightly in each of the models. This affects the representation of the rules in the schema calculus (since it is there that the embedding of outputs is made explicit).

As an illustration consider the angelic model of outputs. As before, when one extracts the simulation rules one finds that the finalisation condition for a upwards simulation places additional constraints on the definition. The nature of angelic outputs also changes the role of a precondition, so that the applicability rule is strengthened in two ways by the finalisation. In terms of the precondition one needs to alter the definition of pre Op to include existential quantification of the after state only (and not the output), since the angelic nature of outputs means that reduction of non-determinism of the output is not possible in a refinement. Hence for an operation Op defined over state space $State$ we define $\mathrm{Pre}\, Op \mathrel{\widehat{=}} \exists\, State' \bullet Op$, and use Pre in place of pre in some of the simulation conditions.

The full set of conditions for an angelic model, with for the sake of illustration a blocking model of preconditions as well, are then:

$$\forall\, CState';\ AState' \bullet T' \wedge CInit \Rightarrow AInit$$
$$\forall\, i : I;\ Input;\ Output;\ AState';\ CState' \bullet T' \wedge COp_i \Rightarrow \exists\, AState \bullet T \wedge AOp_i$$
$$\forall\, CState \bullet \exists\, AState \bullet T \wedge \forall\, Input;\ Output;\ i : I \bullet \mathrm{Pre}\, AOp_i \Rightarrow \mathrm{Pre}\, COp_i$$

9 Conclusions

As we have seen, refinement has progressed a long way since its original definition. Although here we have concentrated on looking at its theoretical foundations, it has also seen industrial application [23, 24], and this in turn has informed the theoretical work we have discussed here.

The basic definition of refinement has produced a number of interesting generalisations, indeed, the sign of a good mathematical theory is when a rich structure arises from a very simple foundation. This has certainly been the case with relational refinement, and thus this is validation of its formulation.

Here we have looked at how it is possible to generalise its definitions by relaxing a number of assumptions in the original definition. First, we considered

notions of observation tied to input and output (IO refinement) or to the state (grey boxes and displays). Second, we looked at the question of conformity and considered how to deal with internal operations (weak refinement) and the decomposition of visible operations (non-atomic refinement). Finally, we returned to the question of observation but this time in the context of generalising finalisations. The latter work makes for some very interesting results and it is clear there is still much work to be done in this area. However, it has already shown how to unify relational and concurrent refinement, producing simulation rules for use in a range of concurrent refinement contexts, and we are sure that more interesting results are set to emerge from work in this area.

References

[1] L. Aceto. Action Refinement in Process Algebras. CUP, London, 1992. 46

[2] E. A. Boiten and J. Derrick. Grey box data refinement. In Grundy et al. [16], pages 45-59. 42

[3] E. A. Boiten and J. Derrick. IO-refinement in Z. In A. Evans, D. J. Duke, and T. Clark, editors, 3rd BCS-FRCS Northern Formal Methods Workshop. Springer-Verlag, September 1998. http://www.ewic.org.uk/. 40, 41

[4] E. A. Boiten and J. Derrick. Liberating data refinement. In R. C. Backhouse and J. N. Oliveira, editors, Mathematics of Program Construction, volume 1837 of Lecture Notes in Computer Science, pages 144-166. Springer-Verlag, 2000. 39

[5] C. Bolton and J. Davies. A Singleton Failures Semantics for Communicating Sequential Processes. Formal Aspects of Computing, 2002. Under consideration. 52

[6] C. Bolton and J. Davies. Refinement in Object-Z and CSP. In M. Butler, L. Petre, and K. Sere, editors, Integrated Formal Methods (IFM 2002), volume 2335 of Lecture Notes in Computer Science, pages 225-244. Springer-Verlag, 2002. 52

[7] J. W. de Bakker and E. P. de Vink. Bisimulation semantics for concurrency with atomicity and action refinement. Fundamenta Informaticae, 20:3-34, 1994. 46

[8] W.-P. de Roever and K. Engelhardt. Data Refinement: Model-Oriented Proof Methods and their Comparison. CUP, 1998. 37

[9] J. Derrick and E. A. Boiten. Non-atomic refinement in Z. volume 1708 of Lecture Notes in Computer Science, pages 1477-1496, Berlin, 1999. Springer-Verlag. 46

[10] J. Derrick and E. A. Boiten. Refinement of state-based systems with internal operations. Technical report, Computing Laboratory, University of Kent, Canterbury, 2000. 44

[11] J. Derrick and E. A. Boiten. Refinement in Z and Object-Z. Springer-Verlag, 2001. 35, 37, 39, 40, 41, 45, 46

[12] J. Derrick, E. A. Boiten, H. Bowman, and M. W. A. Steen. Weak refinement in Z. In J. P. Bowen, M. G. Hinchey, and D. Till, editors, ZUM'97: The Z Formal Specification Notation, volume 1212 of Lecture Notes in Computer Science, pages 369-388. Springer-Verlag, April 1997. 45

[13] J. Derrick, E. A. Boiten, H. Bowman, and M. W. A. Steen. Specifying and Refining Internal Operations in Z. Formal Aspects of Computing, 10:125-159, December 1998. 45, 46

[14] J. Derrick and E. A. Boiten. Refinement of objects and operations in Object-Z. In Formal Methods for Open Object-based Distributed Systems IV, pages 257-277. Kluwer Academic Publishers, 2000. 46

[15] J. Derrick and E. A. Boiten. Unifying concurrent and relational refinement. In REFINE - a FloC'02 FME workshop, Electronic Notes in Theoretical Computer Science. Elsevier, 2002. 52, 53

[16] J. Grundy, M. Schwenke, and T. Vickers, editors. International Refinement Workshop & Formal Methods Pacific '98. Discrete Mathematics and Theoretical Computer Science. Springer-Verlag, Canberra, September 1998.

[17] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data refinement refined. In B. Robinet and R. Wilhelm, editors, Proc. ESOP 86, volume 213, pages 187-196. Springer- Verlag, 1986. 37

[18] He Jifeng. Process refinement. In J. McDermid, editor, The Theory and Practice of Refinement. Butterworths, 1989. 52

[19] M. B. Josephs. A state-based approach to communicating processes. Distributed Computing, 3:9-18, 1988. 52

[20] R. Miarka, E. A. Boiten, and J. Derrick. Guards, preconditions and refinement in Z. In J. P. Bowen, S. Dunne, A. Galloway, and S. King, editors, ZB2000: Formal Specification and Development in Z and B, volume 1878 of Lecture Notes in Computer Science, pages 286-303. Springer-Verlag, September 2000. 36

[21] A. W. Roscoe. The Theory and Practice of Concurrency. International Series in Computer Science. Prentice-Hall, 1998. 52

[22] G. Smith and J. Derrick. Abstract specification in Object-Z and CSP. In ICFEM'02, Lecture Notes in Computer Science. Springer-Verlag, 2002. 54

[23] S. Stepney, D. Cooper, and J. C. P. Woodcock. More powerful data refinement in Z. In J. P. Bowen, A. Fett, and M. G. Hinchey, editors, ZUM'98: The Z Formal Specification Notation, volume 1493 of Lecture Notes in Computer Science, pages 284-307. Springer-Verlag, September 1998. 51, 54

[24] J. C. P. Woodcock. Industrial-strength refinement. In Grundy et al. [16], pages 33-44. 54

[25] J. C. P. Woodcock and J. Davies. Using Z: Specification, Refinement, and Proof. Prentice Hall, 1996. 39

[26] J. C. P. Woodcock and C. C. Morgan. Refinement of state-based concurrent systems. In D. Bjorner, C. A. R. Hoare, and H. Langmaack, editors, VDM'90: VDM and Z!- Formal Methods in Software Development, volume 428 of Lecture Notes in Computer Science. Springer-Verlag, 1990. 52

Partial Updates Exploration II

Yuri Gurevich and Nikolai Tillmann

Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA

Abstract. During one step of a powerful programming, specification or query language, the same mathematical object, like a set or map or sequence, may be modified – in parallel – by different parts of the program. Such partial updates need to be efficiently checked for consistency and integrated. This is the problem of partial updates in a nutshell. In our first paper on the subject we proposed a general solution of the partial-update problem for abstract state machines (ASMs) where the problem is exacerbated by the use of nested submachines; in particular we solved the problem in the cases of counters, sets and maps. Here we propose a more general framework that allows us to solve the problem for sequences and labeled ordered trees. The partial update problem for sequences is related to the problem of collaborative editing.

1 Introduction

Partial updates were introduced into the theory of abstract state machines in [GT01]. There, the parallel composition of operations was identified with order-independent functional composition, so that parallel composition is an abstraction of sequential composition when the order of operations is immaterial. This approach allows one to introduce parallel composition of operations even when simultaneous execution of operations may not be possible.

It turns out that the above framework is too limited. The case in point is that of sequences. Consider, for example, two operations, each of them inserting an element at a different place of a given sequence. The two insertions are compatible and could be executable simultaneously. However the sequential composition of insertions depends on the order in which they are applied. If the right insertion is applied first then the result coincides with that of the simultaneous execution. But if the left insertion is applied first, it offsets the positions to the right of the inserted element, so that the right insertion will insert its element at a wrong place.

In Section 3, we introduce a new framework for partial updates and show how the old framework can be embedded into the new one. We have two case studies: sequences, in Section 4, and labeled ordered trees, in Section 5.

Partial updating of sequences is related to collaborative editing, which is a deep and complicated area subject to much research. Our study of partial updating

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 57–86, 2003.
© Springer-Verlag Berlin Heidelberg 2003

of sequences here is by no means exhaustive but it includes those operations on sequences (or *files*) that we found most natural. The most typical operation is a substitution of the following form: replace the segment of length l of the given file x starting at position p with a sequence s. The segment to be replaced is called the *arena* of the substitution. We don't allow arenas of the form "from p to the end of the sequence" but it would be easy to do so. It is more challenging to extend our study to the case when the replacement sequence s depends on the content of the arena. We only consider a restricted form of this case where the arena contains one position and the replacement sequence contains one element. Such operations are called *alterations*.

The case of labeled ordered trees is essentially the recursive version of the case of sequences.

We presume that the reader is familiar abstract state machines, e.g. with the Lipari Guide [G95] and with the framework of [GT01] though a quick reminder of that framework is provided in Section 2.

2 Preliminaries

2.1 Terminology and Notation

Sequences We consider only finite sequences. Given a sequence x, we write $\#x$ for its length, and x_i denotes the ith element of the sequence, where $i \in \{0, \ldots, \#x - 1\}$. We denote the concatenation of two sequences s and t as $s \cdot t$ or simply st frequently. We denote the empty sequence as ϵ. The term file is used as a synonym for the term sequence.

Multisets We use double curly braces for multisets. For example, $\{\!\{7, 7, 11\}\!\}$ is the multiset that contains 7 with multiplicity 2 and 11 with multiplicity 1 (and 13 with multiplicity 0). The underlying set of a multiset M is its *domain*. The domain of $\{\!\{a, a, b\}\!\}$ is $\{a, b\}$. Then *sum* $A + B$ of multisets A and B is the multiset C such that $\mathtt{dom}(C) = \mathtt{dom}(A) \cup \mathtt{dom}(B)$ and $\mathtt{multiplicity}_C(x) = \mathtt{multiplicity}_A(x) + \mathtt{multiplicity}_B(x)$ for every $x \in \mathtt{dom}(C)$. If B is a submultiset of A then $A - B$ is the multiset C such that $\mathtt{dom}(C) \subseteq \mathtt{dom}(A)$ and $\mathtt{multiplicity}_C(x) = \mathtt{multiplicity}_A(x) - \mathtt{multiplicity}_B(x)$.

Functions The functional composition $(g \circ f)(x) \rightleftharpoons g(f(x))$.

Abstract State Machines The content of a location ℓ of a state X is denoted $\nabla_X \ell$. The value of a term t in a state X is denoted $\nabla_X t$. If $\bar{t}$ is a vector $(t_1, \ldots, v_n)$ of terms then $\nabla_X \bar{t}$ is the vector $(\nabla_X t_1, \ldots, \nabla_X t_n)$.

Concerning the notion of ASM background see [BG00]. Here we assume that the ASM background is closed under tuples and sequences.

2.2 Plain Applicative Algebras

In [GT01], we spoke about the cases of counters, sets and maps. There was a common structure in those and other cases. The time has come to give a name to that structure.

Suppose that we have a type T and an associated type P of unary operations over T such that P is closed under functional composition and contains the identity function *id* over T. The two-sorted algebra A formed by T and P forms a *plain applicative algebra*. T is the *client type* of A, and the unary operations are *particles* of A.

A priori, the same type T can be associated with different particle types. To simplify terminology, we restrict attention to a scenario where, like in [GT01], T uniquely defines the algebra A. The algebra will be denoted $\mathcal{PA}(T)$, and the particles will be called T particles.

We give some particles individual names, e.g.

$$\texttt{identity} : x \mapsto x$$

But typically particles come in families. Each family is the range of a special static function F of positive arity. For example, the overwrite particles

$$\texttt{overwrite}(a) : x \mapsto a$$

form the range of the function `overwrite`. Since the ASM background is closed under tuples, we may assume without loss of generality that every family-generating function F is unary, so that every particle f has the form $F(x)$. It is presumed that the equation $f = F(x)$ has a unique solution. The solution, the *control* of f in F, is produced by a special parameter recovery function $\texttt{ctrl}_F(f)$. We omit the subscript when it is obvious from the context.

2.3 The Old Framework for Partial Updates

For the reader's convenience, we recall the partial-update framework of [GT01] but we use the notion of plain applicative algebra.

A traditional total update has the form $\mathrm{TU}(\ell, v)$ where ℓ is a location and v is a value. A partial update has the form $\mathrm{PU}(\ell, f)$ where ℓ is a location ℓ and f is a particle.

A plain applicative algebra $\mathcal{PA}(T)$ is *apt* if, for every two particles f, g

1. either $(g \circ f)(x) = (f \circ g)(x)$ for all x, that is f and g commute,
2. or $(g \circ f)(x) \neq (f \circ g)(x)$ for all x, that is f and g *malcommute*.

Definition 2.1 A multiset $M = \{\!\{f_1, \ldots, f_n\}\!\}$ of particles is *consistent* if every composition $f_{\pi_1} \circ f_{\pi_2} \circ \cdots \circ f_{\pi_n}$ of its members gives the same particle. Here π ranges over all permutations of $(1, \ldots, n)$. If M is consistent then we write ΠM for the order independent composition. □

Lemma 2.2 ([GT01]) *If $\mathcal{PA}(T)$ then the following are equivalent:*

1. *M is consistent.*
2. *Every two members of M commute.*

Traditionally ASM rules produce update sets. In [GT01], rules produce partial update multisets, that is multisets of partial updates. In particular, the *Partial Update Rule* (*Partial Assignment*) R is a rule

$$f(\bar{t}) \leftarrow t_0$$

where $f(\bar{t})$ is a term of some client type T, and t_0 is a term whose type is that of T particles. In a state X, the rule R produces

$$\tilde{\Delta}_X R \rightleftharpoons \{\!\{\mathrm{PU}(\ell, g)\}\!\}$$

where $\ell = (f, \nabla_X \bar{t})$ and $g = \nabla_X t_0$.

Let ψ be a partial-update multiset, $\mathrm{Loc}(\psi)$ be the set of locations ℓ such that some partial update of ℓ occurs in ψ. Let ℓ range over $\mathrm{Loc}(\psi)$ and define ψ_ℓ to be the multiset $\{\!\{f \mid \mathrm{PU}(\ell, f) \in \psi\}\!\}$. We say a partial-update multiset ψ is *consistent* if every ψ_ℓ is consistent.

We will now explain how to compute traditional update sets from a partial-update multiset. This computation is called *integration*. If ψ is consistent, then for each location $\ell \in \mathrm{Loc}(\psi)$, the integrator computes the product $\mathit{\Pi}\psi_\ell$. The new content of ℓ is $v_\ell \rightleftharpoons (\mathit{\Pi}\psi_\ell)(\nabla_X \ell)$ and the resulting total update is $\mathrm{TU}(\ell, v_\ell)$.

3 A New Framework for Partial Updates

We describe the new framework and examine the relation of the new framework to the one described in [GT01].

3.1 Proper Applicative Algebras

Definition 3.1 An *applicative algebra* A consists of a type T and an associated type P such that every member of P is a unary function over T. The type T contains a special element `undef`. P is closed under functional composition and contains the identity function *id*. In addition, there is an operation $\mathit{\Pi}$ that, given an arbitrary multiset of particles, produces a particle. □

Elements of type T are *elements* of A and members of type P are *particles of* A. `undef` is the *trivial element* of T. T is the *client type* of A. $\mathit{\Pi}M$ is called the *parallel composition* of particles.

Note that the vocabulary of applicative algebras is richer than that of plain applicative algebras. If we want to emphasize that A is non-plain applicative algebra, we will call A a proper applicative algebra.

Again, to simplify notation, we restrict attention to the scenario when different applicative algebras have distinct client types. The applicative algebra with client type T will be denoted $\mathcal{A}(T)$.

Definition 3.2 [Aptness] We say that $A = \mathcal{A}(T)$ is *apt* if the following conditions hold.

A0 A contains a particle that maps every element to `undef`. For simplicity, we identify this *trivial particle* with `undef`, so that $\texttt{undef}(x) = \texttt{undef}$ for all x. Also, $f(\texttt{undef}) = \texttt{undef}$ for any particle f.
A1 For every T particle f, $\Pi\{\!\{f\}\!\} = f$.
A2 For every multiset M of T particles, $\Pi(M + \{\!\{id\}\!\}) = \Pi M$.
A3 $\Pi(M + \{\!\{\texttt{undef}\}\!\}) = \texttt{undef}$ for every multiset M of T particles. □

In particular, if M is empty then $\Pi M = \Pi(M + \{\!\{id\}\!\}) = \Pi\{\!\{id\}\!\} = id$.

Remark 3.3 If the type of interest does not contain `undef`, then the trivial element can always be added in a way such that $f(\texttt{undef}) = \texttt{undef}$ for every particle f. Similarly, if the particle type does not contain `undef`, then the trivial particle can be added in. We don't distinguish between the trivial element of T and the trivial particle over T. One may want to distinguish between the two. This would make no essential difference. □

We say that M is *consistent*, if $\Pi M \neq \texttt{undef}$.

Example 3.4 Let T be a type that contains `undef` and at least one nontrivial element. The collection of T particles contains the identity particle, and the family of `overwrite` particles

$$\texttt{overwrite}(y) : x \mapsto \begin{cases} y & \text{if } x \neq \texttt{undef} \\ \texttt{undef} & \text{otherwise} \end{cases}$$

Note that $\texttt{overwrite}(\texttt{undef})$ is the trivial particle. Define the product g of a multiset of particles as follows. (1) If M consists of some number (possibly zero) of identity particles then $g = id$. (2) If M contains `undef` or two distinct overwrite particles, then $g = \texttt{undef}$. (3) If neither (1) nor (2) holds then M consists of identity particles and at least one copy of an overwrite particle f; in this case $g = f$. It is easy to see that the resulting applicative algebra is apt. A multiset of particles is inconsistent if and only if it contains `undef` or two distinct overwrite particles. □

3.2 Properties of Apt Applicative Algebras

Requirements A0–A3 seem pretty trivial. We didn't need additional requirements at this point, but there are natural candidates for such additional requirements.

Consider an apt applicative algebra $\mathcal{A}(T)$ and define $f \times g = \Pi\{\!\{f, g\}\!\}$.

Definition 3.5 $\mathcal{A}(T)$ is *associative* if the binary operation $f \times g$ is associative. □

Definition 3.6 $\mathcal{A}(T)$ is *distributive* if $\Pi(M_1 + \cdots + M_n) = \Pi\{\!\{\Pi M_1, \ldots, \Pi M_n\}\!\}$ for all multisets $M_1, \ldots, M_n$ of T particles. $\mathcal{A}(T)$ is *conditionally distributive* if $\Pi(M_1 + \cdots + M_n) = \Pi\{\!\{\Pi M_1, \ldots, \Pi M_n\}\!\}$ provided that $\Pi(M_1 + \cdots + M_n) \neq \texttt{undef}$. □

If $\mathcal{A}(T)$ is associative then the binary operation $f \times g$ gives rise to a multiset operation $\{\!\{f_1, \ldots, f_n\}\!\} \mapsto f_1 \times \cdots \times f_n$. However, this multiset operation does not necessarily coincides with the multiset operation of $\mathcal{A}(T)$.

Lemma 3.7 *$\mathcal{A}(T)$ is distributive if and only if it is associative and $\Pi\{\!\{f_1, \ldots, f_n\}\!\} = f_1 \times \cdots \times f_n$ for any particles $f_1, \ldots, f_n$.*

The proof is obvious. Distributivity does not follow from associativity alone. Indeed, consider Example 3.4 and redefine the clause (3) as follows: If neither (1) nor (2) holds then M consists of identity particles and some number $k \geq 1$ of copies of an overwrite particle f; if $k \leq 2$ then $g = f$, otherwise $g = \texttt{undef}$.

Definition 3.8 $\mathcal{A}(T)$ is *locally consistent* if the following holds for every element x: for every multiset M of particles, if $\Pi M \neq \texttt{undef}$ and $f(x) \neq \texttt{undef}$ for all f in M then $(\Pi M)(x) \neq \texttt{undef}$. $\mathcal{A}(T)$ is *pairwise locally consistent* if the following holds for every element x: for every two T particles f, g, if $f \times g \neq \texttt{undef}$ and $f(x) \neq \texttt{undef}$ and $g(x) \neq \texttt{undef}$ then $(f \times g)(x) \neq \texttt{undef}$. □

Lemma 3.9 *Suppose that $\mathcal{A}(T)$ is distributive. If it is pairwise locally consistent, then it is locally consistent.*

Proof By induction on n, we prove the following statement $S(n)$: for any x of type T and any particles $f_1, \ldots, f_n$, if $\Pi\{\!\{f_1, \ldots, f_n\}\!\} \neq \texttt{undef}$ and every $f_i(x) \neq \texttt{undef}$ then $(\Pi M)(x) \neq \texttt{undef}$. The case $n \leq 1$ it is trivial. We assume $S(n)$ and prove $S(n+1)$. So suppose that $(\Pi\{\!\{f_1, \ldots, f_{n+1}\}\!\} \neq \texttt{undef}$ and every $f_i(x) \neq \texttt{undef}$ for all $i \in \{1, \ldots, n+1\}$. By the induction hypothesis, $(\Pi\{\!\{f_1, \ldots, f_n\}\!\})(x) \neq \texttt{undef}$. By pairwise local consistency, $(\Pi\{\!\{f_1, \ldots, f_n\}\!\} \times f_{n+1})(x) \neq \texttt{undef}$. By distributivity, $(\Pi\{\!\{f_1, \ldots, f_n, f_{n+1}\}\!\})(x) \neq \texttt{undef}$. □

Definition 3.10 $\mathcal{A}(T)$ is *pervasive up* if the following holds for any particles $f_1, \ldots, f_n$: if $f_i \times f_j \neq \texttt{undef}$ for all $i < j$ then $\Pi\{\!\{f_1, \ldots, f_n\}\!\} \neq \texttt{undef}$. $\mathcal{A}(T)$ is *pervasive down* if the following condition holds: if M_2 is a multiset of particles and M_1 is a submultiset of M_2 then $\Pi M_2 \neq \texttt{undef}$ implies $\Pi M_1 \neq \texttt{undef}$. □

Example 3.11 We construct an distributive, apt applicative algebra $\mathcal{A}(T)$ that is not pervasive up. The client type T is the powerset of the set $\{a, b, c\}$. We identify the empty set with `undef`. We say that an operation f over T is a *quasi identity* if $f(x) = x$ or $f(x) = \texttt{undef}$ for every element x of type T. For each $S \subseteq T - \{\texttt{undef}\}$, there is a quasi identity f_S such that $f_S(x) = x$ if x in S and $f_S(x) = \texttt{undef}$ otherwise; every quasi identity is some f_S. Stipulate that the type of particles consists of the quasi identities. We have the trivial particle $\texttt{undef} = f_\emptyset$ and $id = f_S$ where $S = T - \{\texttt{undef}\}$. For every multiset $M = \{\!\{f_{S_1}, \ldots, f_{S_n}\}\!\}$ of particles, define $\Pi M = f_T$ where $T = S_1 \cap \cdots \cap S_n$. It is easy to see that this applicative algebra is apt and distributive. Let $S = \{a, b\}, T = \{b, c\}, U = \{c, a\}$, then f_S, f_T, f_U are particles such that $f_S \times f_T \neq \texttt{undef}$, $f_T \times f_U \neq \texttt{undef}$ and $f_U \times f_S \neq \texttt{undef}$, but $\Pi\{\!\{f_S, f_T, f_U\}\!\} = \texttt{undef}$. □

Lemma 3.12 *Let $\mathcal{A}(T)$ be a conditionally distributive, apt applicative algebra. Then $\mathcal{A}(T)$ is pervasive down.*

Proof Let M_2 be a multiset of T particles such that $\Pi M_2 \neq \texttt{undef}$, and let M_1 be a submultiset of M_2. Then $\Pi M_2 = (\Pi M_1) \times (\Pi(M_2 - M_1))$. It follows that $\Pi M_1 \neq \texttt{undef}$. □

Corollary 3.13 *Let $\mathcal{A}(T)$ be an apt applicative algebra that is pervasive up and down. For every multiset M of particles, the following are equivalent:*

1. *M is consistent.*
2. *M is pairwise consistent.*

3.3 Turning Plain Applicative Algebras into Proper Applicative Algebras

Let $A_0 = \mathcal{PA}(T)$ be an apt plain applicative algebra with client type T. Suppose that A_0 does not contain `undef`; it is not an element of T and it is not a T particle. In this case, extend A_0 by means of `undef` in such a way that **A0** holds. In [GT01], the parallel composition $\Pi\{\!\{f_1, \ldots, f_n\}\!\}$ of particles was defined only in the case when the functional composition of the n particles is order independent; in that case $\Pi\{\!\{f_1, \ldots, f_n\}\!\} = f_1 \circ \cdots \circ f_n$. Extend this definition to arbitrary multisets M of particles by setting $\Pi M = \texttt{undef}$ in the case when M contains `undef` or else there are different orderings of M that give different functional compositions. It is easy to see that the requirements **A0**–**A3** are satisfied so that the resulting proper applicative algebra A is apt.

Now suppose that A contains `undef` either as an element or as a particle. In this case, redefine the denotation of `undef`. The element of A that played the role of `undef` does not do that anymore. Now we have the scenario in the previous paragraph.

The resulting apt, proper applicative algebra A has many good properties.

- A is locally consistent, as only the `undef` particle maps nontrivial elements to `undef`.
- A is conditionally distributive, because, if $\Pi(M_1 + \cdots + M_n) \neq \texttt{undef}$ for multisets $M_1, \ldots, M_n$ of particles, then $(M_1 + \cdots + M_n)$ is consistent and the composition of all particles in $(M_1 + \cdots + M_n)$ is order independent.
- A is pervasive up and down. Both properties follow immediately from Lemma 2.2.

3.4 Revisiting Counters, Sets and Maps

We show that the derived (proper) applicative algebras over sets and maps are distributive but the derived applicative algebra over counters is not even associative.

Let $A = \mathcal{A}(T)$ be an apt, proper applicative algebra derived from an apt, plain applicative algebra A_0 as described in Subsection 3.3. We use the notions of commuting and malcommuting that apply to particles of A_0 and thus to nontrivial particles of A when restricted to nontrivial elements. Thus, every two nontrivial particles f and g of A either commute or malcommute. In the first case, $f \times g \neq \texttt{undef}$, and in the second case, $f \times g = \texttt{undef}$. In either case, $f \circ g \neq \texttt{undef}$.

Lemma 3.14 *The following properties are equivalent:*

1. *A is distributive.*
2. *In A_0, if f, g malcommute and g, h commute, then f and $(g \circ h)$ malcommute.*

Proof
1 ⇒ 2. Assume *1*. Then A is associative. Suppose that f, g malcommute and g, h commute. Then

$$f \times (g \circ h) = f \times (g \times h) = (f \times g) \times h = undef \times h = \texttt{undef}$$

It follows that f and $g \circ h$ malcommute.

2 ⇒ 1. Assume *2*. First we prove that A is associative. Given T particles f, g, h, we check that $(f \times g) \times h = f \times (g \times h)$. This is obvious if one of the particles is `undef`. So suppose that all the particles f, g, h are original A_0 particles. Several cases arise.

- f, g commute and g, h commute and f, h commute. So every two members of $M \rightleftharpoons \{\!\{f, g, h\}\!\}$ commute. Then we know by Lemma 2.2 that M is consistent and thus $(f \circ g) \circ h = h \circ (f \circ g) \neq \texttt{undef}$ and $(g \circ h) \circ f = f \circ (g \circ h) \neq \texttt{undef}$. This means that $f \circ g = f \times g$ commutes with h and $g \circ h = g \times h$ commutes with f. Thus $(f \times g) \times h = (f \circ g) \circ h = f \circ (g \circ h) = f \times (g \times h)$.
- f, g commute and g, h commute and f, h malcommute. For every nontrivial element y of type T, $(f \circ h)(y) \neq (h \circ f)(y)$. In particular, for every nontrivial x, $(f \circ h)(gx) \neq (h \circ f)(gx)$. Hence $(f \circ (h \circ g))(x) = (f \circ h)(gx) \neq (h \circ f)(gx) = (h \circ f \circ g)(x) = (h \circ g \circ f)(x) = ((h \circ g) \circ f)(x)$ for all nontrivial x of type T, so that f and $g \circ h = h \circ g$ malcommute. By symmetry, $f \circ g$ and h malcommute. Thus $f \times (g \times h) = \texttt{undef} = (f \times g) \times h$.
- f, g commute and g, h malcommute. Then $f \times (g \times h) = f \times \texttt{undef} = \texttt{undef}$. By *2*, $(f \times g) \times h = \texttt{undef}$.
- f, g malcommute and g, h commute. Then $(f \times g) \times h = \texttt{undef} \times h = \texttt{undef}$. By *2*, $f \times (g \times h) = \texttt{undef}$.
- f, g malcommute and g, h malcommute. Then $(f \times g) \times h = \texttt{undef} \times h = \texttt{undef}$ and $f \times (g \times h) = f \times \texttt{undef} = \texttt{undef}$.

Thus the parallel composition is associative. By Lemma 3.7, it suffices to prove that $\Pi\{\!\{f_1, \ldots, f_n\}\!\} = f_1 \times \cdots \times f_n$ for any particles $f_1, \ldots, f_n$. Without loss of generality, all n particles are nontrivial. If the n particles pairwise commute then the equality follows from the definition of parallel composition. So assume that not all n particles pairwise commute.

It suffices to prove the following: $g_1 \times \cdots \times g_m = \texttt{undef}$ if g_1 and g_m malcommute. We prove this by induction on m. The case $m \leq 2$ is obvious; use the definition of parallel composition. So assume that $m > 2$. If g_{m-1} malcommutes with g_m then $g_{m-1} \times g_m = \texttt{undef}$ and therefore $g_1 \times \cdots \times g_m = \texttt{undef}$. So assume that g_{m-1} commutes with g_m. Then swap g_{m-1} and g_m and use the induction hypthesis. □

Lemma 3.15 *Let A_0 be the apt, plain applicative algebra over maps as defined in [GT01]. Then the derived apt, proper algebra A over maps is distributive.*

Proof By the lemma, it suffices to check that A_0 satisfies the condition *2*. We use the terminology of [GT01]. Let f, g, h be ranked transformers such that f, g malcommute and g, h commute. We prove that f and $g \circ h$ malcommute.

We say that a transformer f is *map-valued*, if $f(m)$ is a map for every point m. Note that if f is not map-valued then it is of rank 0 and so $f(x) = a$ for all points x and for some non-map point a. Recall that a map-valued transformer f has a controller c with associated transformers c_x such that $c_x(mx) = (fm)(x)$ for all points m and x. Theorem 11.12 in [GT01] implies that two map-valued transformers f, g with controllers c, d respectively commute if and only if c_x and d_x commute for every point x.

The proof goes by induction on $\text{rank}(g)$. Suppose g has rank zero. Recall that a zero-ranked transformer is a constant function, so $g \circ h = g$ as g, h commute.

As f, g malcommute, so do f and $g = g \circ h$. So let g be of rank > 0. Then g is map-valued. As $g \circ h = h \circ g$, h returns a map at least at some points. It follows that h is map-valued. Indeed, if $h(x) = a$ and a is not a map for some x, then $\text{rank}(h) = 0$ and so $h(x) = a$ for all points x which is impossible.

If f is not map-valued, then f and $g \circ h$ malcommute because $f \circ (g \circ h)$ is not a map while $(g \circ h) \circ f$ is a map. So assume f is map-valued. Let c, d, e be the controllers of f, g, h respectively. As f, g malcommute, there is a particular x such that c_x, d_x malcommute. Similarly, all d_y, e_y commute because g, h commute; in particular d_x and e_x commute. We know that $\text{rank}(d_x) < \text{rank}(g)$. By induction hypothesis, c_x and $d_x \circ e_x$ malcommute. Hence f and $g \circ h$ malcommute. □

Lemma 3.16 *Let A_0 be the apt, plain applicative algebra over sets as defined in [GT01]. Then the derived apt, proper applicative algebra A over sets is distributive.*

Proof Again, it suffices to check that A_0 satisfies the condition *2*. Let f, g, h be set particles. Suppose g is an overwrite particle. Then $g \circ h = g$. As f, g malcommute, so do f and $g = g \circ h$. So let g be an insert-and-remove operation, and so every $((g \circ h) \circ f)(x)$ is a set. So assume every $(f \circ (g \circ h))(x)$ is a set. f could be an insert-and-remove particle or an overwrite particle that produces some set a; in the second case, we can see f as a generalized insert-and-remove particle: given a set s, it removes all elements in s that do not belong to a and then inserts all elements of a. As f, g malcommute, there are two cases:

Case 1: There is an element x that f inserts and g removes. Since g, h commute, h cannot insert that x. Hence $x \in (f \circ (g \circ h))(x)$ and $x \notin ((g \circ h) \circ f)(x)$. Therefore f and $g \circ h$ malcommute.

Case 2: There is an element x that f removes and g inserts. Because g, h commute, h cannot remove that x. Hence $x \notin (f \circ (g \circ h))(x)$ and $x \in ((g \circ h) \circ f)(x)$. Therefore f and $g \circ h$ malcommute. □

Let A_0 be the apt, plain applicative algebra over counters as defined in [GT01]. Then the derived apt, proper applicative algebra A over sets is not distributive and not even associative. A counterexample is given by the following three particles: f is an overwrite, g is an increment by one and h is a decrement by one.

3.5 The Product of Applicative Algebras

We define the product of two apt applicative algebras. In the similar way, one can define the product of several apt applicative algebras. One can also define a sum of apt applicative algebras; see the end of this subsection.

Assume that $A_1 = \mathcal{A}(T_1)$ and $A_2 = \mathcal{A}(T_2)$ are apt applicative algebras. Let T be the direct product $T_1 \times T_2$, except that we declare (x, y) to be the trivial element of T if x or y is the trivial element in T or U respectively.

The particles over T are `alter` particles working componentwise.

$$\texttt{alter}(f,g) : (x,y) \mapsto (f(x), g(y))$$

where f is a T_1 particle and g is a T_2 particle. If f or g is the trivial particle of T_1 or T_2 respectively then $\texttt{alter}(f,g)$ is the trivial particle particle of T. If f and g are the identity particles of T_1 and T_2 respectively then $\texttt{alter}(f,g)$ is the identity particle of T. This gives us a new applicative algebra $A = A_1 \times A_2 = \mathcal{A}(T)$. The parallel composition of T particles is defined componentwise as well: compose the first components of the given T particles and separately compose their second components. It is easy to check that T is apt. If both A_1 and A_2 are locally consistent the so is T. A multiset of T particles is consistent if and only the multiset of the fist components is consistent in A_1 and the multiset of the second components is consistent in A_2.

Remark 3.17 One can define the sum (or disjoint union) $A = A_1 + A_2$ of applicative algebras $A_1 = \mathcal{A}(T_1)$ and $A_2 = \mathcal{A}(T_2)$ in a very natural way. The summand types T_1 and T_2 are presumed to have only the trivial element in common. The new particles are T_1 particles extended to produce `undef` at any element of T_2 and T_2 particles extended to produce `undef` at any element of T_1 and the identity particle. We leave the details as an exercise. Again, the construction can be extended to several summands. □

3.6 Integration of Partial Updates for Apt Algebras

The integration process is similar to that in [GT01] but somewhat different.

Let ψ be a partial-update multiset, $\mathrm{Loc}(\psi)$ be the set of locations ℓ such that some partial update of ℓ occurs in ψ, and ℓ range over $\mathrm{Loc}(\psi)$. Define ψ_ℓ to be the multiset $\{\!\{f \mid \mathrm{PU}(\ell, f) \in \psi\}\!\}$. According to [GT01], ψ is consistent if every ψ_ℓ is consistent, and inconsistent ψ cannot be ever integrated. The situation is different in the present framework.

Definition 3.18 ψ is *consistent at a state* X if every $\Pi(\psi_\ell)(\nabla_X \ell) \neq \texttt{undef}$. □

We assume that ψ is consistent at a state X and explain how to integrate ψ. For each location $\ell \in \mathrm{Loc}(\psi)$, the integrator computes the product $f_\ell \rightleftharpoons \Pi\psi_\ell$. If $v_\ell \rightleftharpoons \nabla_X \ell$ then $v'_\ell \rightleftharpoons f_\ell(v_\ell)$ is the new content of ℓ. The resulting total update is $\mathrm{TU}(\ell, v'_\ell)$.

3.7 The Cost of Consistency Check

Given a multiset $M = \{\!\{f_1, \ldots, f_n\}\!\}$ of particles, what is the cost to decide whether it is consistent? In the case of a plain applicative algebras, apriori, one compares the compositions of all $n!$ possible permutations of $f_1, \ldots, f_n$. By

Lemma 2.2, in the case of plain applicative algebra (also, by Corollary 3.13, in the case of an apt, proper applicative algebra that is pervasive up and down), only pairwise commutativity of the elements of M must be checked to establish consistency. A naive pairwise checking involves $O(n^2)$ comparisons.

In general, a parallel composition function for multisets may be hard to describe. Given a distributive, pervasive up, apt, proper applicative algebra, we can decide the consistency of a multiset M and compute the product ΠM using $O(n)$ binary parallel compositions. This is based on the following observations that follows from Corollary 3.13.

Corollary 3.19 *Let $\mathcal{A}(T)$ be a distributive, pervasive up, apt applicative algebra. Let M be a multiset of particles.*

- *If M is empty, then M is trivially consistent and $\Pi M = id$.*
- *Let $M = M' + \{\!\{f\}\!\}$ for some f. Then M is consistent iff $\Pi M'$ is consistent and $(\Pi M') \times f \neq \texttt{undef}$. If M is consistent then $\Pi M = (\Pi M') \times f$.*

There are important natural examples of apt applicative algebras that are distributive and pervasive up, e.g. the applicative algebras over sets and maps; see Subsection 3.4.

The applicative algebra over counters is not distributed (or even associative), but there is a very efficient algorithm for parallel composition (and thus for checking consistency). We use the terminology of [GT01]. Consider a multiset M of counter particles. If M contains `undef`, return `undef`. Suppose that M does not contain `undef`. If there is at least one non-zero increment and an overwrite then return `undef`. If you have two distinct overwrites, return `undef`. Otherwise compute the functional composition of particles in M in any order.

The applicative algebra over sequences is not associative either. But again there is an efficient algorithm for parallel composition; see the end of Section 4.

3.8 Reporting

The reporting process is similar to that in [GT01] with the following difference. There it was not defined what happens if the partial-update multiset computed by a submachine is inconsistent. Now, the use of `undef` allows us to report any partial-update multiset computed by the submachine.

We are going to define the sequential composition of partial-update multisets ψ and ψ' for apt applicative algebras. The intention is that ψ is executed first and then ψ' is executed in the state resulting from the execution of ψ'.

Definition 3.20 Let ψ and ψ' be two partial-update multisets and let $L = \mathrm{Loc}(\psi) \cup \mathrm{Loc}(\psi')$. For each $\ell \in L$, let $g_\ell = \Pi\{\!\{f \mid \mathrm{PU}(\ell, f) \in \psi\}\!\}$ and $g'_\ell = \Pi\{\!\{f \mid \mathrm{PU}(\ell, f) \in \psi'\}\!\}$. Then

$$\psi \ \texttt{seq} \ \psi' \rightleftharpoons \{\!\{\mathrm{PU}(\ell, g'_\ell \circ g_\ell) \ \mid \ \ell \in L\}\!\} \qquad \square$$

Corollary 3.21 *Let ψ and ψ' be two partial-update multisets and X be a state such that ψ is locally consistent in X and ψ' is locally consistent in Y, where Y is X plus the result of the integration of ψ. Then ψ `seq` ψ' is locally consistent in X.*

This corollary guarantees that if local consistency holds in each step of a submachine, then the reported partial-update multiset of the whole submachine will be locally consistent in the current state of the parent machine. This property for the sequential composition corresponds to the local consistency property for parallel composition.

Finally, we repeat the definition of sequential composition of rules. It is identical to the one given in [GT01].

Definition 3.22

$$\tilde{\Delta}_X(P \texttt{ seq } Q) \rightleftharpoons \tilde{\Delta}_X P \texttt{ seq } \tilde{\Delta}_Y Q$$

where Y is X plus the result of the integration of $\tilde{\Delta}_X P$. □

4 Example: Sequences

We consider sequences of elements of some fixed type T, construct an applicative algebra over sequences and analyze it.

4.1 Motivation

Sequences form one of the most basic data structures of computing. Text processors, like Emacs or Microsoft Word, treat the given text as a sequence of characters (a file) and allow you to perform basic operations on the file: insert a string, delete a part of the file, alter a part of a file. Two or more people people may independently change two copies of the same file and later merge their changes; this merging is essentially the integration of two partial updates of the file. Each person may sequentially perform multiple basic operations; done programmatically, this gives rise to a submachine. In addition, each person may employ assistants to do independent changes, and the assistants may use assistants of their own. If all this is done programmatically, we have nested submachines. This is how collaborative editing relates to partial updates, integration and submachines.

In this connection we looked at

- The UNIX diff and patch commands, and
- The merge-and-compare feature of the Microsoft Word.

The diff command takes two files: the original and a derived document. A file is viewed as a sequence of lines. The command computes the difference of the derived document from the original one in terms of inserted, deleted or altered segments of lines. The result is stored as a "patchfile". The patch command applies such a patchfile to a document and is fairly tolerant to what document the patchfile can be applied to. It does not need to be the original document, as long as it is sufficiently similar to it. Similar usually means that there is a so-called horizon of two unmodified lines around every insertion, deletion and alteration described in the patchfile. Because of this tolerance, one can try to apply several patchfiles, one after another, to a document. If two-line horizon rule not respected, then patches are rejected. A rejection may happen even if the modifications are consistent but happen to be too close.

The "Merge and Compare" feature of Microsoft Word is more sophisticated and complicated. When a document is modified, not only the final result is saved but also the relative changes, the "deltas", are saved in terms of insertions and deletions of segments of characters and alterations of the style (font style, paragraph style, etc.) of character ranges. Starting from the original document, one can merge several sets of relative changes of the original document. Deletions and insertions are highlighted. The Word relies on a detailed description of the relative changes and does not require horizons. Even contradicting modifications can be merged; the modifications are applied sequentially.

We will describe a new framework for partial updates of sequences, quite natural in our opinion. Admittedly our sequence-modification operations can be made richer. But, within the framework, our algorithms handle parallel, arbitrarily nested modifications of the same sequence and detects contradictions.

4.2 Natural Operations on Sequences

The most natural operations on sequences seem to be insertion, deletion and alteration. The sequence being modified will be often called a *file*.

What is a *position* in a file? Notice that a position is not necessarily uniquely defined by the element in that position; for example the element b appears at two positions in the file abb. It is natural to number the positions say from left to right and identify a position with its number. Imagine now that you insert an element into the file. Do you have to renumber the positions and thus change their identities? We are not trying to solve the metaphysical problem what position really is. We will get around that problem.

If you want to insert a sequence s into a file x of length n, there are $n+1$ places to insert s: at the very beginning of x, between any two adjacent positions, at the end of x. We will use the term *place* in this sense.

Using the terms position and place, we can describe the three basic modifications of a given file.

- Insert a sequence s into a given place of the given file.
- Delete the part of the file that is given by a segment of positions.
- Alter the element of type T at a given position by means of a given T particle.

Insertion and deletion can be combined into a single substitution operation. Then there are only two basic families of operations: substitutions and alterations.

4.3 The Setup

Fix an arbitrary apt application algebra $A = \mathcal{A}(T)$ such that the type T contains at least two nontrivial elements. Let S be the type of sequences of elements in $T - \{\texttt{undef}\}$ enriched with the trivial element `undef`. We construct an apt applicative algebra $SEQ(A) \rightleftharpoons \mathcal{A}(S)$.

Recall that, if x is a sequence, then $\#x$ is the length of a sequence x and the elements of x are $x_0, \ldots, x_{\#x-1}$. If $x = \texttt{undef}$ define $\#x = \texttt{undef}$; and if x is a sequence but i isn't a valid position of x, define $x_i = \texttt{undef}$. (Note that the ASM background should be rich enough to allow us to have the $\#x$ and x_i functions.)

4.4 Particles

We will define the sequence particles in two steps. We first define the family of so-called *prime* particles, which represent single substitutions, alterations, and position requirements. (A position requirement enforces that a certain position is present in the file. This prime particle may result from the sequential composition of an insertion and a deletion, or from the composition of several alter particles, as we will see later.) Then we define complex sequence particles from such prime particles.

As $\mathcal{A}(T)$ is apt, a T particle α is in general a partial function, i.e. $\alpha(x)$ may equal `undef` for some x. In this connection, we introduce guards. A *guard* for a T particle α is the predicate $\texttt{guard}_\alpha(x) \iff \alpha(x) \neq \texttt{undef}$ over T. We write `true` for the trivial guard that always holds and `false` for the unsatisfiable guard. The use the abstraction of guards will become apparent for the sequential composition of particles. To illustrate the need for guards, consider a simple substitution g and an alteration f such that f operates on a position i that g replaces. Intuitively, the sequential composition $g \circ f$ absorbs the alteration. But one has to be careful: $g = g \circ f$ does not hold if the T particle α of the alteration f is partial. Then a sequence x exists such that $\alpha(x_i) = \texttt{undef}$. As `undef` represents an undefined element that signals an error condition, we want to propagate this information such that $f(x) = \texttt{undef}$. Thus g cannot be identified with $g \circ f$, as the simple substitution g does not contain the information about the condition that must hold for f to be defined on x. We will use guards to preserve this information.

Prime Substitution Particles The control of a *prime substitution particle* f is given by a natural number p (the *anchor position* of f), a sequence σ of satisfiable guards and a replacement sequence s. Symbolically

$$f = \mathtt{sub}(p, \sigma, s)$$

The point of requiring the guards to be satisfiable is to avoid that the trivial particle `undef` is a prime substitution particle. We require only that every single guard is satisfiable but this implies that there is a file where all guards are satisfied; the reason is that different guards are applied at different positions.

We define the *arena* of f in abstraction from the input sequence (the file) x. Then we define when x is *long enough* for f. Then we define the *arena of f in x* provided that x is long enough for f.

Case 1: σ is empty. In this case, $\mathtt{arena}(f)$ consists of the place π immediately before the anchor position p. A file x is long enough for f if $\mathtt{length}(x) \geq p$. Suppose that x is long enough for f. If $p = 0$ then π is the place at the beginning of x. If $0 < p < \mathtt{length}(x)$ then π is the place between the positions $p-1$ and p in x. And if $0 < p = \mathtt{length}(x)$ then π is the place at the end of x.

Case 2: σ is not empty. In this case, $\mathtt{arena}(f)$ consists of positions $p, \ldots, p + \#\sigma - 1$. A file x is long enough for f if $\mathtt{length}(x) \geq p + \#\sigma - 1$. Suppose that x is long enough for f. Then arena of f in x is the segment of x containing the positions $p, \ldots, p + \#\sigma - 1$ of x. This is an initial segment of x if $p = 0$ and a final segment of x if $p + \#\sigma = \mathtt{length}(x)$.

Now we explain how f modifies a file x. If $x = \mathtt{undef}$ then $f(x)\mathtt{undef}$. So assume x is a nontrivial sequence. If x is not long enough for f or one of the guards in σ fails then $f(x) = \mathtt{undef}$. Assume that x is long enough for f and all guards in σ hold.

Case 1: $\#\sigma = 0$. Then f is an insertion. It inserts s at the unique place in $\mathtt{arena}(f)$ that is immediately before the position p.

Case 2: $\#\sigma > 0$ and s is empty. Then f is a deletion. It removes the segment v of x given by the arena of f. If $x = uvw$ then $f(x) = uw$.

Case 3: $\#\sigma > 0$ and s is nonempety. Then f is a proper substitution. It replaces the segment v of x given by arena of f with s. If $x = uvw$ then $f(x) = usw$.

Two substitution particles $f = \mathtt{sub}(p, \sigma, s)$ and $g = \mathtt{sub}(q, \tau, t)$ are *adjacent* if their domains are adjacent, that is if $p + \#\sigma = q$ or $q + \#\tau = p$. In particular, two insertions are adjacent if their arenas coincide.

Lemma 4.1 *Suppose that substitution particles $f = \mathtt{sub}(p, \sigma, s)$ and $g = \mathtt{sub}(q, \tau, t)$ are adjacent and $q \leq p$. Let $h \rightleftharpoons \mathtt{sub}(q, \tau \cdot \sigma, t \cdot s)$. Then, for every file x, we have $h(x) = g(f(x))$.*

The proof is obvious.

Prime Alter Particles We call a T particle $\alpha \neq \texttt{undef}$ *quasi constant* if there is an element b of type T such that $\alpha(a) = b$ whenever $\texttt{guard}_\alpha(a)$ holds.

The control of a *prime alter particle* g is given by a positive integer p, the *anchor position* of g, and by a T particle α satisfying the following conditions.

- $\alpha \neq \texttt{undef}$ (so that `undef` is not a prime alter particle).
- α is not the identity particle (so that no prime alter particle is a prime position requirement particle).
- α is not quasi constant (so that no prime alter particle is a prime substitution particle).

We write

$$g = \texttt{alter}(p, \alpha)$$

The *arena* of g consists of the single position p. A file x is *long enough* for g if $\texttt{length}(x) > p$, so that position p exists in x.

Let x be a file. If $x = \texttt{undef}$ then $f(x)\texttt{undef}$. So assume x is a nontrivial sequence. If x is not long enough for f or it is long enough for f but $\alpha(x_p) = \texttt{undef}$ then $g(x) = \texttt{undef}$. Otherwise $g(x)$ is obtained from x by replacing x_p with $\alpha(x_p)$.

Prime Position Requirement Particles We call position requirement particles *require* particles. The control of a *prime require particle* h is given by a positive integer p, the position that must be present in a file.

We write

$$h = \texttt{require}(p)$$

The *arena* of h consists of the single position p. A file x is *long enough* for h if $\texttt{length}(x) > p$, so that position p exists in x.

Let x be a file. If $x = \texttt{undef}$ then $f(x)\texttt{undef}$. So assume x is a nontrivial sequence. If x is not long enough for h, then $h(x) = \texttt{undef}$. Otherwise $h(x) = x$.

Remark 4.2 The require particle is could also be seen as a prime particle with an internal identity particle, or as a substitution with no guards and an empty replacement sequence. But require particles are treated very differently in the following, which is why we do not identify them with particular alterations or substitutions. □

Properties of Prime Particles We refer to prime substitution, alter, and require particles as prime particles. If I and J are sets of natural numbers, we say that I is to the left of J, symbolically $I < J$, if $i < j$ for every $i \in I$ and every $j \in J$. We extend this definition to the situation when I and J are the arenas of prime particles f and g respectively. The only interesting new cases are when f or g or both are insertions.

Case 1: I is the arena of an insertion particle $f = \texttt{sub}(p, \epsilon, s)$ and J is a set of natural numbers. Define $I < J$ if $p \leq j$ for all $j \in J$.

Case 2: I is a set of natural numbers and J is the arena of an insertion particle $g = \texttt{sub}(q, \epsilon, t)$. Define $I < J$ if $i < q$ for all $i \in I$.

Case 3: $I = \texttt{arena}(\texttt{sub}(p, \epsilon, s))$ and $J = \texttt{arena}(\texttt{sub}(q, \epsilon, t))$. Define $I < J$ if $p < q$.

Let f and g be two prime particles. f is *to the left of* g, symbolically $f < g$, if $\texttt{arena}(f)$ is to the left of $\texttt{arena}(g)$. This gives a partial order on prime particles.

We say that prime particles $f_1, \ldots, f_n$ form a *chain* if every $f_i < f_{i+1}$ holds.

Every set of particles where every two particles are ordered forms a chain.

We say that a require particle $g = \texttt{require}(q)$ is *essential* for a collection C of prime particles if $f < g$ for every $f \in C - \{g\}$. A chain $f_1, \ldots, f_n$ of prime particles is *normal* if

- for no i, the particles f_i and f_{i+1} are adjacent prime substitution particles, and
- no f_i with $i < n$ is a require particle (so that there are no inessential require particles).

Lemma 4.3 *Every chain $f_1, \ldots, f_n$ of prime particles can be normalized, that is transformed into a normal chain $g_1, \ldots, g_m$ such that $\Pi\{\!\{f_1, \ldots, f_n\}\!\} = \Pi\{\!\{g_1, \ldots, g_m\}\!\}$.*

Proof Repeatedly use Lemma 4.1 and remove inessential prime require particles. □

Sequence Particles and their Prime Factors A *nontrivial sequence particle* f is given by a normal chain of prime particles called the *prime factors* of f. The normal chain of prime factors can be empty in which case f is the identity particle. In addition to such particles with a prime factor representation, the trivial particle `undef` is also a sequence particle.

Let f be a nontrivial sequence particle. To apply f to a file x, apply all the prime factors of f to x simultaneously. Since the arenas of the prime factors are disjoint, this definition is sound.

Can this parallel application of prime factors be reduced to the usual functional composition? Yes, but only if we apply the prime factors in the right-to-left order.

Corollary 4.4 *Let $f_1, \ldots, f_n$ be the chain of prime factors of a sequence particle f. Then*

$$f = f_1 \circ f_2 \circ \cdots \circ f_{n-1} \circ f_n$$

where $\circ$ is the standard functional composition.

Remark 4.5 We do not define a distinct family of `overwrite` sequence particles. In order to overwrite a file x of length n by a sequence s, we can use $\mathtt{sub}(0, \sigma, s)$ where σ is the sequence $\mathtt{true} \cdots \mathtt{true}$ of length n. □

The following lemma states that the prime decompositions, that is prime factor representation of a nontrivial sequence particle, is unique.

Lemma 4.6 (Prime Decomposition) *Every nontrivial sequence particle has a unique prime decomposition.*

Proof We start with a couple of useful definitions. Call a file x *appropriate* for a sequence particle f if $f(x) \neq \mathtt{undef}$. By the definition of nontrivial sequence particles, every nontrivial sequence particle has appropriate files. Note that every end extension of a file appropriate for f is appropriate for f.

If x is a file of length $> p$ and a is an element of type T, then $x(p/a)$ is the file obtained from x by setting the value at position p to a.

For every prime sequence particle g, there is a number $\delta(d)$ such that $\mathtt{length}(gx) - \mathtt{length}(x) = \delta(g)$ for every file x appropriate for g. If g is a prime substitution particle and $\mathtt{ctrl}\, g = (p, \sigma, s)$ then $\delta(g) = \mathtt{length}(s) - \mathtt{length}(\sigma)$; and if g is an alter particle then $\delta(g) = 0$.

Let f be a nontrivial sequence particle. The rest of the proof splits in two parts. In the first part, we analyze an arbitrary prime decomposition of f; by the definition of nontrivial sequence particles, f has a prime decomposition. In the second part, we reconstruct a prime decomposition of f. The reconstruction is deterministic and independent of the decomposition choice in the first part. It uses only the properties of f itself. This shows that there is only one prime decomposition.

Part 1: Analysis. Let $D = (f_1, \ldots, f_m)$ be a decomposition of f, and A_i be the arena of f_i. Only f_m could be a require particle. For every natural number p define $\xi(p) = p + \sum_{i \in I} \delta(f_i)$ where I consists of induces i such that A_i is to the left of p.

Suppose that p does not belong to any A_i. Then p satisfies the following condition.

C0 There exists a unique natural number q such that, for every file x of length $> p$ that is appropriate for f, we have the following. For every nontrivial element a of type T, the file $x' \rightleftharpoons x(p/a)$ is appropriate for f and $\mathtt{length}(fx') > q$ and $(fx')_q = x'_p$.

Suppose that p is the anchor position of an alter particle f_j so that the control of f_j has the form (p, α). For every file x appropriate for f, we have $(fx)_{\xi p} = \alpha(x_p)$. By the definition of prime alter particles, α is not identity and not quasi constant. The p satisfies the following condition.

C1 There exists a unique natural number q such that, for every file x appropriate for f, we have the following.
- There exist two distinct nontrivial elements a, b of type T such that $x(p/a)$ and $x(p/b)$ are appropriate for f and $(f(x(p/a)))_q \neq (f(x(p/b)))_q$, and
- There exists a nontrivial element a of type T such that either $x(p/a)$ is not appropriate for f or else $(f(x(p/a)))_q \neq a$.

Suppose that p belongs to the arena of a substitution particle f_j so that the control of f_j has the form (p, σ, s). Then p satisfies the following condition.

C2 Let x be any file appropriate for f and a be any nontrivial element of type T. If $x(p/a)$ is appropriate for f then $f(x(p/a)) = f(x)$.

Suppose that p is the anchor of a require particle f_j. Then $j = m$, and p satisfies the following condition.

C3 p fails to satisfy **C0**–**C2**, and there exists a file of length $p+1$ appropriate for f, and no file of length p is appropriate for f.

It is easy to see that every natural number p satisfies exactly one of the conditions the conditions **C0**–**C3**.

Part 2: Reconstruction. We reconstruct a decomposition of f. Every natural number p satisfies exactly one of the conditions **C0**–**C3**.

Case 0: p satisfies **C0**. Then p does not belong to the arena of any prime factor, and the number q is $\xi(p)$.

Case 1: p satisfies **C1**. Then p is the anchor of some prime alter factor g and $q = \xi(p)$. The control (p, α) of g is uniquely determined. To find any $\alpha(a)$, construct a file $x'' = x(p/a)$ and compute $(fx'')_{\xi p}$.

Case 2: p satisfies **C2**. Then p belongs to the arena of some prime substitution factor. This way we can determine the arenas of all prime substitution factors of f that are not pure insertions.

Case 3: p satisfies **C3**. Then p is the anchor of a require particle. .

Let A be the arena of a substitution factor g of f that is not a pure insertion. The control (p, σ, s) of g is determined uniquely. The anchor position $p = \min A$. Let $q = 1 + \max(A)$ and let x be any file appropriate for f of length $> q$. To determine any $\sigma_i(a)$, construct a file $x' = x(p + i/a)$ and check whether x' is appropriate for f. To determine s, we need to find the appropriate segment y of $f(x)$. Since substitution factors cannot have adjacent arenas, $\xi(q)$ is defined. Similarly, if $p > 0$ then $\xi(p-1)$ is defined. The end position of y is $\xi(q) - 1$. If $p > 0$ then the initial position of y is $\xi(p-1) + 1$; otherwise the initial position of y is 0.

It remains to determine the prime insertion factors. Let p be any natural number and let x be a file appropriate for f of length $> p$. First suppose that $p = 0$.

In this case, f has a prime insertion factor $\mathtt{sub}(0, \epsilon, s)$ if and only if $\xi(0) > 0$ in which case s is the initial segment of $f(x)$ to the left of position $\xi(0)$. Second suppose that $p > 0$. In this case, f has a prime insertion factor $\mathtt{sub}(p, \epsilon, s)$ if and only if $\xi(p-1) + 1 < \xi(p)$ and s is the segment of $f(x)$ between the positions $\xi(p-1) + 1$ and $\xi(p)$. □

4.5 Parallel Composition

We define the parallel composition of sequence particles in two steps: first we define the parallel composition of prime particles, and only then we define the parallel composition of arbitrary sequence particles.

Definition 4.7 Let M be a multiset $\{\!\{f_1, \ldots, f_m\}\!\}$ of prime particles. Define the parallel composition of M to equal `undef`, symbolically $\Pi M \rightleftharpoons \mathtt{undef}$, if any of the following conditions holds:

- There are $i \neq j$ such that f_i is a substitution particle and f_j is either substitution or alter particle and the pair f_i, f_j is unordered.
- M includes a maximal multiset of prime alter particles $\{\!\{g_1, \ldots, g_n\}\!\}$ with the same anchor and with internal particles $\alpha_1, \ldots, \alpha_n$ respectively such that $\Pi\{\!\{\alpha_1, \ldots, \alpha_n\}\!\} = \mathtt{undef}$.

Suppose that none of the above conditions holds. Then the prime factors of ΠM are obtained as follows from M.

First fuse multiple alter particles with the same anchor. For each maximal submultiset M' of M that consists of alter particles with the same anchor, do the following. Let $M' = \{\!\{g_1, \ldots, g_n\}\!\}$, p be the common anchor and β be the parallel product of the n internal particles.

- If β is the identity, then replace M' with $\mathtt{require}(p)$.
- If β is quasi constant, then there exists a unique $b \neq \mathtt{undef}$ such that $\beta(a) = b$ for some a. In this case, replace M' with $\mathtt{sub}(p, \mathtt{guard}_\beta, b)$.
- Otherwise replace M' with $\mathtt{alter}(p, \beta)$.

Second remove superfluous require particles. Keep only the require particle (if there is one like that) that is to the right of every other particle in the domain of M. Furthermore, keep only one copy of it.

The remaining particles form a chain. Finally, normalize that chain. The resulting normal chain is the prime decomposition of ΠM. □

We now define the parallel composition of a multiset M of arbitrary sequence particles.

Definition 4.8 Let M be a multiset of sequence particles. If `undef` is in M, then $\Pi M \rightleftharpoons$ `undef`. Otherwise each particle f in M has a prime decomposition M_f. Take the sum $M' \rightleftharpoons \sum_{f \in M} M_f$. The parallel composition $\Pi M'$ of prime particles has been defined above. Set $\Pi M \rightleftharpoons \Pi M'$. □

By construction of the parallel composition, the following corollary is obvious:

Corollary 4.9 *Let $f_1, \ldots, f_n$ be the prime factors of a sequence particle f. Then the following holds: $f = \Pi\{\!\{f_1, \ldots, f_n\}\!\}$*

Corollary 4.4 gives rise to the following corollary:

Corollary 4.10 *Let $f_1, \ldots, f_n$ be a chain of prime particles and $f = \Pi\{\!\{f_1, \ldots, f_n\}\!\}$. For every file x, $f(x) \neq$ `undef` iff $f_1(x) \neq$ `undef`, $\ldots, f_n(x) \neq$ `undef`.*

4.6 Sequential Composition

Given two sequence particles f, g, we show that $g \circ f$ is again a sequence particle, i.e. it is either `undef` or it has a prime decomposition. First we assume that f and g are prime, and then we do the general case.

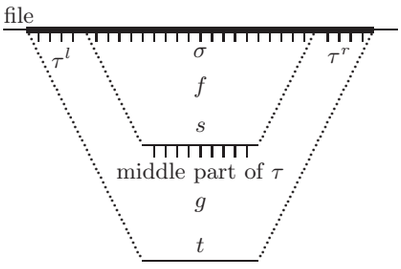
Sequential Composition of two Prime Particles There is an algorithm that decides whether $g \circ f \neq$ `undef`, and if so finds a prime decomposition of $g \circ f$. In fact, the prime decomposition consists of at most two prime particles. If there is only one prime particle, it will be called h_1. If there are two prime particles, they will be $h_2 < h_1$. We explain how algorithm works.

Case 1: f and g are prime substitution particles with $\texttt{ctrl}\, f = (p, \sigma, s)$ and $\texttt{ctrl}\, g = (q, \tau, t)$ respectively. This case splits into a number of subcases.

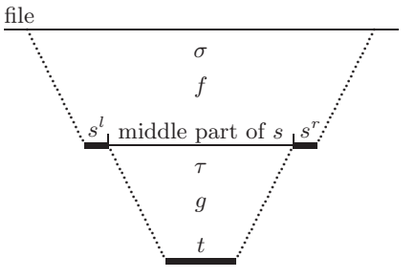
1.1: $p + \#\sigma < q$, so f is to the left of g and f, g are not adjacent. Construct $h_2 = f$, $h_1 = \texttt{sub}(q - \delta(f), \tau, t)$.
1.2: $p + \#\sigma = q$ and $\#\sigma + \#\tau \neq 0$, so that f is to the left of g but f and g are adjacent. Construct $h_1 = \texttt{sub}(p, \sigma \cdot \tau, s \cdot t)$.
1.3: $p = q$ and $\#\sigma = \#\tau = 0$, so that both f and g are pure insertions with the same anchor. Construct $h_1 = \texttt{sub}(p, \epsilon, t \cdot s)$.
1.4: $q + \#\tau < p$, so that g is to the left of f and f, g are not adjacent. Construct $h_2 = g$ and $h_1 = f$.
1.5: $q + \#\tau = p$, so that either both f, g are pure insertions with the same anchor or else g is to the left of f and f, g are adjacent. In either case, we combine them to a single substitution. Construct $h_1 = \texttt{sub}(q, \tau \cdot \sigma, t \cdot s)$.
1.6: None of the above, so that $(p + \#\sigma > q) \wedge (q + \#\tau > p)$.

By assumption, f, g are not trivial, i.e. there exist files x, y such that $f(x) \neq \texttt{undef}$ and $g(y) \neq \texttt{undef}$. Is there always a file x such that $g(f(x)) \neq \texttt{undef}$? To answer this question, examine $f(x)$. Some elements of $f(x)$ are inserted by f into x, and the other elements are inherited from x. In order to have $g(f(x)) \neq \texttt{undef}$, we can choose the inherited elements in such a way that they satisfy the appropriate guards in τ; recall that the guards in τ are satisfiable. But a guard τ_i may fail at an element inserted by f (the element s_{q+i-p} to be exact) in which case $g \circ f \rightleftharpoons \texttt{undef}$. So let us assume that all τ guards operating on elements inserted by f hold.

If f is an insertion and g is a deletion such that $p = q$ and $\#s = \#\tau$, then g deletes exactly the sequence inserted by f. Hence $g \circ f$ does not modify a file. If $p = 0$, then $g \circ f$ is the identity. Otherwise $g \circ f$ does not modify the elements of a file but it is not identity; the result is $h_1 \rightleftharpoons \texttt{require}(p-1)$. So let us assume that $g \circ f$ is neither identity nor a require particle.



Composing Guards



Composing Replacement Sequences

Now we can fuse f and g into a single substitution particle h_1; the two pictures illustrate the fusion. The anchor position of the fused substitution is $r \rightleftharpoons \min\{p, q\}$. The guard sequence of h_1 is $\upsilon \rightleftharpoons \tau^l \cdot \sigma \cdot \tau^r$ where τ^l and τ^r are as follows. τ^l is nonempty only in the case $q < p$; in that case it consists of the guards $\tau_0 \cdots \tau_{p-q-1}$. τ^r is nonempty only in the case that $q + \#\tau > p + \#s$; in that case it consists of the guards $\tau_{p+\#s-q} \cdots \tau_{\#\tau-1}$. The replacement sequence of h_1 is $u \rightleftharpoons s^l \cdot t \cdot s^r$ where s^l and s^r are as follows. s^l is nonempty only if $p < q$; in that case it is the subsequence $s_0 \cdots s_{q-p-1}$ of s. s^r is nonempty only if $p + \#s > q + \#\tau$; in which case it is the subsequence $s_{q+\#\tau-p} \cdots s_{\#s-1}$ of s.

Case 2: f and g are prime alter particles with $\texttt{ctrl}\, f = (p, \alpha)$ and $\texttt{ctrl}\, g = (q, \beta)$. This case splits into a number of subcases.

2.1: $p < q$, so f is to the left of g. Construct $h_2 = f$ and $h_1 = g$.
2.2: $p = q$. Let $\gamma \rightleftharpoons \beta \circ \alpha$.
  2.2.1: $\gamma = \texttt{undef}$. Then $g \circ f \rightleftharpoons \texttt{undef}$.
  2.2.2: γ is the identity. Then construct $h_1 = \texttt{require}(p)$.
  2.2.3: γ is quasi constant. Then there exists $b = \alpha(a) \neq \texttt{undef}$ for some a. Construct $h_1 = \texttt{sub}(p, \texttt{guard}_\gamma, b)$.

2.2.4: None of the above. We can combine them into a single prime alter factor. Construct $h_1 = \texttt{alter}(p, \gamma)$.

2.3: $p > q$, so g is to the left of f. Construct $h_2 = g$ and $h_1 = f$.

Case 3: f is a prime substitution and g is a prime alter particle with $\texttt{ctrl}\, f = (p, \sigma, s)$ and $\texttt{ctrl}\, g = (q, \beta)$. This case splits into a number of subcases.

3.1: $p + \#s \leq q$, so f is to the left of g. Construct $h_2 = f$ and $h_1 = \texttt{alter}(q - \delta(f), \beta)$.

3.2: $q < p$, so g is to the left of f. Construct $h_2 = g$ and $h_1 = f$.

3.3: None of the above. i.e. $(p + \#s > q) \wedge (q \geq p)$. This means that g affects an element s_{q-p} that is part of the replacement sequence of f. If $\beta(s_{q-p}) = \texttt{undef}$, then $g \circ f \rightleftharpoons \texttt{undef}$. Otherwise we can combine f and g into one substitution: Construct $h_1 = \texttt{sub}(p, \sigma, s_0 \cdots s_{q-p-1} \cdot \beta(s_{q-p}) \cdot s_{p-q+1} \cdots s_{\#s-1})$.

Case 4: f is a prime alter and g is a prime substitution particle with $\texttt{ctrl}\, f = (p, \alpha)$ and $\texttt{ctrl}\, g = (q, \tau, t)$. This case splits into a number of subcases.

4.1: $p < q$, so f is to the left of g. Construct $h_2 = f$, and $h_1 = g$.

4.2: $q + \#\tau \leq p$, so g is to the left of f. Construct $h_2 = g$ and $h_1 = f$.

4.3: None of the above, i.e. $(p \geq q) \wedge (q + \#\tau > p)$. This means that f affects an element that g will replace. So the composition $g \circ f$ is almost equivalent to g; it only differs in that we have to compose the particle α with the corresponding guard in τ. If $\tau_{p-q} \circ \alpha = \texttt{false}$ then $g \circ f \rightleftharpoons \texttt{undef}$. Otherwise construct $h_1 = \texttt{sub}(p, \tau_0 \cdots \tau_{p-q-1} \cdot (\tau_{p-q} \circ \alpha) \cdot \tau_{p-q+1} \cdots \tau_{\#\tau-1}, t)$.

Case 5: f is a prime substitution and g is a require particle with $\texttt{ctrl}\, f = (p, \sigma, s)$ and $\texttt{ctrl}\, g = (q)$. This case splits into a number of subcases.

5.1: $p + \#s \leq q$, so f is to the left of g, and g is essential. Construct $h_2 = f$, and $h_1 = \texttt{require}(q - \delta(f))$.

5.2: Otherwise g is inessential. Construct $h_1 = f$.

Case 6: Suppose f is a prime alter and g is a require particle with $\texttt{ctrl}\, f = (p, \alpha)$ and $\texttt{ctrl}\, g = (q)$. This case splits into a number of subcases.

6.1: $p < q$, so f is to the left of g, and g is essential. Construct $h_2 = f$, and $h_1 = g$.

6.2: Otherwise g is inessential. Construct $h_1 = f$.

Case 7: f is a prime require and g a prime substitution and g particle with $\texttt{ctrl}\, f = (p)$ and $\texttt{ctrl}\, g = (q, \tau, t)$. This case splits into a number of subcases.

7.1: $p \geq q + \#\tau$, so g is to the left of f, and f is essential. Construct $h_2 = g$, and f.

7.2: Otherwise f is inessential. Construct $h_1 = g$.

Case 8: Suppose f is a prime require and g is an alter particle with $\texttt{ctrl}\, f = (p)$ and $\texttt{ctrl}\, g = (q, \beta)$. This case splits into a number of subcases.

8.1: $p > q$, so g is to the left of g, and f is essential. Construct $h_2 = g$ and $h_1 = f$.
8.2: Otherwise f is inessential. Construct $h_1 = g$.

Case 9: Suppose f is a prime require and g is a require alter particle with $\texttt{ctrl}\, f = (p)$ and $\texttt{ctrl}\, g = (q)$. Construct $h_1 = \texttt{require}(\max\{p, q\})$.

Corollary 4.11 *Let f, g, h be three prime particles such that $g \circ f$ is a prime particle $h_{\min} \rightleftharpoons h_1$ or has a prime decomposition $h_{\min} \rightleftharpoons h_2 < h_1$. If h is to the left of both f and g, then h is to the left of $h_{\min}$.*

Sequential Composition in General

Lemma 4.12 *For any prime particles $f_l, \ldots, f_1$, there are only two cases:*

- $f_l \circ \cdots \circ f_1 = \texttt{undef}$.
- *There is a chain $g_m, \ldots, g_1$ of prime particles with $m \leq l$ such that $f_l \circ \cdots \circ f_1 = g_m \circ \cdots \circ g_1$. Furthermore, if $m \geq 1$ then, for any prime particle h, if h is to the left of every f_i then it is to the left of g_m.*

Proof We prove the lemma by induction on l. For $l \leq 1$, it is trivial. Assume that the lemma holds for some $l \geq 1$. Let f_{l+1} be a prime particle. If $f_l \circ \cdots \circ f_1 = \texttt{undef}$, then $f_{l+1} \circ f_l \circ \cdots \circ f_l = \texttt{undef}$. So assume $f_l \circ \cdots \circ f_1 \neq \texttt{undef}$. By induction hypothesis, $g_m, \ldots, g_1$ is a chain of prime particles such that $f_l \circ \cdots \circ f_1 = g_m \circ \cdots \circ g_1$. If $m = 0$, then f_{l+1} is the desired chain. Otherwise the application of the above algorithm to $f_{l+1} \circ g_m$ yields one of the following cases.

- $f_{l+1} \circ g_m = \texttt{undef}$. In this case, $f_{l+1} \circ f_l \circ \cdots \circ f_1 = f_{l+1} \circ g_m \circ \cdots \circ g_1 = \texttt{undef}$ and we are done.
- $f_{l+1} \circ g_m$ is the identity (by case 1.6). In this case, $f_{l+1} \circ \cdots \circ f_1 = g_{m-1} \circ \cdots \circ g_1$ and thus $g_{m-1}, \ldots, g_1$ is the desired chain.
  If $m \geq 2$ and a prime particle h is to the left of all f_i, $1 \leq i \leq l+1$, then by the induction hypothesis, it is to the left of g_m and therefore to the left of g_{m-1}.
- $f_{l+1} \circ g_m = \bar{g}_m$ for some prime particle $\bar{g}_m$. In this case, $f_{l+1} \circ f_l \circ \cdots \circ f_1 = f_{l+1} \circ g_m \circ \cdots \circ g_1 = \bar{g}_m \circ g_{m-1} \circ \cdots \circ g_1$. As $m \leq l$, the induction hypothesis implies that there exists a chain $h_n, \ldots, h_1$ of prime particles such that $\bar{g}_m \circ g_{m-1} \circ \cdots \circ g_1 = h_n \circ \cdots \circ h_1$ and $n \leq m < l + 1$.
  If $n \geq 1$ and h is to the left of all f_i then – by the induction hypothesis – h is to the left of g_m. By the Corollary above, h is to the left of $\bar{g}_m$, so that h is to the left of $\bar{g}_m, g_{m-1}, \ldots, g_1$. Applying the induction hypothesis again, we have that h is to the left of h_n.

– $f_{l+1} \circ g_m = \bar{g}_{m+1} \circ \bar{g}_m$ where $\bar{g}_{m+1} < \bar{g}_m$ are prime particles.
If $\bar{g}_{m+1}, \bar{g}_m$ arise from cases 1.4, 2.3, 3.2, 4.2, 7.1, 8.1, then $f_{l+1} = \bar{g}_{m+1} < \bar{g}_m = g_m$. Thus $\bar{g}_{m+1}, g_m, \ldots, g_1$ is the desired chain such that $f_{l+1} \circ \cdots \circ f_1 = \bar{g}_{m+1} \circ g_m \circ \cdots \circ g_1$. For any prime particle h that is to the left of all f_i, we have $h < f_{l+1} = \bar{g}_{m+1}$.
Otherwise $\bar{g}_{m+1}, \bar{g}_m$ arise from cases 1.1, 2.1, 3.1, 4.1, 6.1. In all these cases, $\bar{g}_{m+1} = g_m < \bar{g}_m$. By induction hypothesis, there is a chain $h_n, \ldots, h_1$ such that $\bar{g}_m \circ g_{m-1} \circ \cdots \circ g_1 = h_n \circ \cdots \circ h_1$ and $n \leq m$. If $n = 0$, the desired chain consists of $\bar{g}_{m+1}$ only. Suppose that $n \geq 1$. Since $\bar{g}_{m+1}$ (that is g_m) is to the left of $\bar{g}_m$ and all g_i with $i < m$, by induction hypothesis, we have that $\bar{g}_{m+1} < h_n$. Therefore the desired chain is $\bar{g}_{m+1}, h_n, \ldots, h_1$ with length $n + 1 \leq m + 1 \leq l + 1$. Indeed,

$$\begin{aligned} & \bar{g}_{m+1} \circ h_n \circ \cdots \circ h_1 \\ = \; & \bar{g}_{m+1} \circ \bar{g}_m \circ g_{m-1} \circ \cdots \circ g_1 \\ = \; & f_{l+1} \circ g_m \circ g_{m-1} \circ \cdots \circ g_1 \\ = \; & f_{l+1} \circ f_l \circ f_{l-1} \circ \cdots \circ f_1 \end{aligned}$$

For any prime particle h that is to the left of all f_i, it follows by the induction hypothesis that $h < g_m = \bar{g}_{m+1}$. □

Using the last lemma, we can now proof that sequence particles are closed under sequential composition.

Lemma 4.13 *Let f and g be sequence particles. Then the composition $g \circ f$ is either a trivial or yields a nontrivial sequence particle.*

Proof If either f or g is trivial, then $g \circ f =$ `undef`. So assume that f and g are nontrivial. Let $f_l, \ldots, f_1$ be the chain of prime factors of f and let $g_m, \ldots, g_1$ be the chain of prime factors of g. We know that $g \circ f = (\Pi\{\!\{g_1, \ldots, g_m\}\!\}) \circ (\Pi\{\!\{f_1, \ldots, f_l\}\!\}) = g_m \circ \cdots \circ g_1 \circ f_l \circ \cdots \circ f_1$. According to Lemma 4.12 there either exists a chain of prime particles $h_n, \ldots, h_1$ such that $g \circ f = h_n \circ \cdots \circ h_1$ or $g \circ f =$ `undef`. If $g \circ f =$ `undef`, then we are done. So assume there is an equivalent chain. Using Lemma 4.3, we know that there exists an equivalent normal chain. □

4.7 Aptness

We have shown that the collection of sequence particles is closed under functional composition and under parallel composition. The applicative algebra $SEQ(A)$ over sequences fulfills all requirements for aptness.

A0 `undef` is an element of the client type S of $SEQ(A)$ by the definition of S. `undef` and id are sequence particles. (Recall that the nontrivial sequence particle with the empty prime decomposition is the identity particle.) $\texttt{undef}(x) = f(\texttt{undef}) =$ `undef` for all sequences x and all sequence particles f by the definition of $SEQ(A)$.

A1 The equality $\Pi\{\!\{f\}\!\} = f$ follows from the definition of Π.
A2 The equality $\Pi(M + \{\!\{id\}\!\}) = \Pi M$ follows from the fact that the identity particle has the empty prime decomposition.
A3 The equality $\Pi(M + \{\!\{\texttt{undef}\}\!\}) = \texttt{undef}$ follows from the definition of Π.

4.8 Properties of the Applicative Algebra over Sequences

The parallel composition is not associative. Indeed, let $f \times g \rightleftharpoons \Pi\{\!\{f, g\}\!\}$ and consider the following example: $h = \texttt{sub}(0, \texttt{true}, c)$, $g = \texttt{sub}(0, \epsilon, b)$, $f = \texttt{sub}(0, \epsilon, a)$. We have $(h \times g) \times f = \texttt{sub}(0, \texttt{true}, bc) \times f = \texttt{sub}(0, \texttt{true}, abc)$, but $h \times (g \times f) = h \times \texttt{undef} = \texttt{undef}$.

If $A = \mathcal{A}(T)$ is associative then the only reason that $SEQ(A)$ is not associative is that insertions can be adjacent to proper substitutions or alterations. If we forbid such constellations then $SEQ(A)$ becomes associative.

Lemma 4.14 *Assume that $A = \mathcal{A}(T)$ is locally consistent. Then $SEQ(A)$ is locally consistent.*

Proof Let x be a sequence and M be a multiset of sequence particles. Suppose that $\Pi M \neq \texttt{undef}$ and $f(x) \neq \texttt{undef}$ for every f in M. We show that $\Pi M(x) \neq \texttt{undef}$.

Since $\Pi M \neq \texttt{undef}$, M does not contain the trivial `undef` particle, and so each particle in M has a prime decomposition. Let M' be the multiset of all prime factors of all members of M so that $\Pi M = \Pi M'$. Clearly $f(x) \neq \texttt{undef}$ for all f in M'.

The prime decomposition of ΠM is a normal chain that results from the integration of M': the fusion of alter prime particles with the same anchor, from the fusion of adjacent prime substitute particles, and from the removal of superfluous prime require particles. It suffices to examine each prime particle f in the prime decomposition of ΠM and show that $f(x) \neq \texttt{undef}$.

- $f = \texttt{alter}(p, \alpha)$. This prime particle is the result of the fusion of a multiset M'' of one or more prime alter particles in M'. By the local consistency of A, we have that $f(x) \neq \texttt{undef}$.
- $f = \texttt{sub}(p, \sigma, s)$. This prime particle is the result of the fusion of a series M'' of adjacent prime substitute particles in M'. Formally M'' is a set that contains at least one member. A member g of M'' may have resulted from the fusion a multiset M''' of prime alter particles in M'; in such a case $g(x) \neq \texttt{undef}$ by the local consistency of A. Since adjacent prime substitute particles can be applied simultaneously, we have $f(x) \neq \texttt{undef}$.
- $f = \texttt{require}(p)$. This prime particle belongs to M' or is the result of a fusion of alter particles with the same anchor p. In either case $f(x) \neq \texttt{undef}$. □

Analogous to the above lemma, it is easy to see by the definition of the parallel composition that $SEQ(A)$ is conditionally distributive, if A is conditionally distributive. Similarly, $SEQ(A)$ is pervasive up (down), if A is pervasive up (down).

4.9 Consistency Check

We sketch an efficient algorithm to check the consistency of a multiset M of sequence particles. First compute the multiset M' of the prime factors of all particles in M. Then sort particles in M' according to their anchor positions; sort particles with the same anchor position arbitrarily. Walk through the sorted list left to right and check that every two neighbors in the list are compatible.

5 Example: Labeled Ordered Trees

By *ordered tree* we mean a tree where the children of each node are linearly ordered, and thus form a sequence. (This "horizontal" order on children should be clearly distinguished from the "vertical" partial order of the tree itself.) A *labeled ordered tree* is an ordered tree where each node has a label. What are natural operations on labeled ordered trees? We can think of

- insertion of a labeled ordered subtree,
- deletion of a subtree,
- alteration of a label

For brevity, labeled ordered trees are called *trees* in the rest of this section; no other trees will be considered.

5.1 Construction

Fix an apt applicative algebra $A = \mathcal{A}(L)$ where L is a type with at least two nontrivial elements. Think about elements of type L as labels for trees. Consider a recursive equation

$$X = A \times SEQ(X)$$

where X ranges over applicative algebras. The least fixed point of this equation will be called $LOT(A)$, an allusion to *L*abeled *O*rdered *T*rees over A.

Let T be the client type of $LOT(A)$, so that $LOT(A) = \mathcal{A}(T)$. Let us analyze T. By the definition of the product of applicative algebras in Subsection 3.5, an arbitrary nontrivial element x of type T has the form (ℓ, s) where ℓ is a label and s is a sequence of elements of type T. If s is empty, so that x is a tree with an ℓ-labeled root that has no children, assign rank 0 to x. If $\#s = n > 0$ and every element s_i of s has been assigned a rank r_i, assign rank $1 + \max_i r_i$ to x.

Assign rank 0 to `undef`. This way, every element of type T gets a rank. It is easy to see that every nontrivial element x of type T is a labeled ordered tree of depth equal to the rank of x.

Next we analyze T particles. By the definition of the product of applicative algebras in Subsection 3.5, an arbitrary nontrivial T particle f has the form (α, g) where α is a nontrivial L particle and g is a nontrivial sequence particle. If g does not have any prime alter factors, assign rank 0 to f. In particular the identity particle over T gets rank 0. If g does have prime alter particles and every prime alter particle h has been assigned a rank r_h, assign rank $1 + \max_h r_h$ to f. Assign rank 0 to `undef`. This way, every particle over T gets a rank.

It is easy to check that $LOT(A)$ is apt.

5.2 Insertions, Deletions, Alterations

We start by defining an auxiliary function $\texttt{node}(x, s)$ that, given a tree x and a sequence s of natural numbers produces the subtree of x at position s. If $s = []$, that is if s is empty, then $\texttt{node}(x, s)$ is x. If $s = [i]$, that is if s has just one member i, then $\texttt{node}(x, s)$ is the child number i of x (where the leftmost child is number 0) provided that x has at least $i + 1$ children; otherwise $\texttt{node}(x, s) = \texttt{undef}$. If $s = [i, j]$ and $\texttt{node}(x, [i]) \neq \texttt{undef}$ and $\texttt{node}(x, [i])$ has a child y of number j then $\texttt{node}(x, s) = y$; otherwise $\texttt{node}(x, s) = \texttt{undef}$. And so on.

In the rest of this subsection, we describe the three natural operations over trees mentioned above – insertion, deletion and label alteration – more precisely and check that they are legitimate T particles.

Insertion The insertion operation $f = \texttt{insert}(s, y)$ takes two parameters: a nonempty sequence s of natural numbers and a tree y. Since s is nonempty it has the form $t \cdot [p]$. Let x be a nontrivial tree. If $\texttt{node}(x, t) = \texttt{undef}$ or else $\texttt{node}(x, t) \neq \texttt{undef}$ and $p > 0$ and $\texttt{node}(x, t)$ has less then p children then $f(x) = \texttt{undef}$. So assume that $\texttt{node}(x, t)$ is a nontrivial tree with at least p children. If $\texttt{node}(x, t)$ has exactly p children so that the last child is number $p - 1$, then attach y as the pth child of $\texttt{node}(x, t)$. If $\texttt{node}(x, t)$ has more than p children, then insert y between the child number $p - 1$ and the child number p of $\texttt{node}(x, t)$ so that y becomes the child number p of $\texttt{node}(t, x)$.

Recall that, according to the definition of the product of application algebras in Subsection 3.5, the only particles of $LOT(A)$ are alter particles. We illustrate that $\texttt{insert}(s, y)$ is a T particle on the example of $s = [p, q]$.

$$\texttt{insert}([p, q], y) = \texttt{alter}(id, \texttt{alter}(p, \texttt{alter}(id, \texttt{sub}(q, \epsilon, y)))).$$

Deletion The deletion operation $f = \texttt{delete}(s)$ takes one parameter: a nonempty sequence s of natural numbers. Since s is nonempty it has the

form $t \cdot [p]$. Let x be a nontrivial tree. If $\texttt{node}(x, t) = \texttt{undef}$ or else $\texttt{node}(x, t)$ is nontrivial but does not have the child number p then $f(x) = \texttt{undef}$. Otherwise $f(x)$ is obtained from x by deleting the pth child of $\texttt{node}(x, t)$.

Again, we illustrate that $\texttt{delete}(s)$ is a T particle on the example of $s = [p, q]$. In this case,

$$\texttt{delete}([p, q]) = \texttt{alter}(id, \texttt{alter}(p, \texttt{alter}(id, \texttt{sub}(q, \texttt{true}, \epsilon)))).$$

Label Alteration The label operation $f = \texttt{la}(s, \alpha)$ takes two parameters: a possibly empty sequence s of natural numbers and a nontrivial label particle $\alpha \neq id$. Let x be a nontrivial tree. If $\texttt{node}(x, s) = \texttt{undef}$ or if $\texttt{node}(x, s)$ is nontrivial with some label ℓ but $\alpha(\ell) = \texttt{undef}$ then $f(x) = \texttt{undef}$. Otherwise $f(x)$ is obtained from x by replacing the label ℓ of $\texttt{node}(x, s)$ with $\alpha(\ell)$.

Again, we illustrate that $\texttt{la}(s, \alpha)$ is a T particle on the example of $s = [p, q]$. In this case,

$$\texttt{la}([p, q], \alpha) = \texttt{alter}(id, \texttt{alter}(p, \texttt{alter}(id, \texttt{alter}(q, \texttt{alter}(\alpha, id))))).$$

References

[BG00] Andreas Blass and Yuri Gurevich, "Background, reserve, and Gandy machines," in *Computer Science Logic*, P. Clote and H. Schwichtenberg, eds., Springer Lecture Notes in Computer Science 1862 (2000), 1–17. 58

[BG02] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," ACM Transactions on Computation Logic, to appear.

[ASM] ASM Michigan web page, `http://www.eecs.umich.edu/gasm`, maintained by Jim Huggins.

[FSE] Foundations of Software Engineering, Microsoft Research, web page, `http://research.microsoft.com/fse/`.

[G95] Yuri Gurevich, "Evolving algebra 1993: Lipari guide," in *Specification and Validation Methods*, E. Börger, ed., Oxford Uni. Press (1995) 9–36. 58

[GT01] Yuri Gurevich and Nikolai Tillmann, "Partial Updates: Exploration," in *Journal of Universal Computer Science (J.UCS)*, 7(11):917–951, 2001. 57, 58, 59, 60, 63, 65, 66, 67, 68, 69

Experiments with Test Case Generation and Runtime Analysis

Cyrille Artho[1], Doron Drusinksy[2], Allen Goldberg[3], Klaus Havelund[3], Mike Lowry[4], Corina Pasareanu[3], Grigore Roşu[5], and Willem Visser[6]

[1] Computer Systems Institute, ETH Zurich, CH-8092 Zurich, Switzerland
[2] Naval Postgraduate School, Monterey, California, USA, and Time Rover, Inc, Cupertino, California, USA
[3] Kestrel Technology, NASA Ames Research Center, Moffett Field, California USA
[4] NASA Ames Research Center, Moffett Field, California USA
[5] Department of Computer Science, University of Illinois at Urbana-Champaign
[6] RIACS, NASA Ames Research Center, Moffett Field, California USA

Abstract. Software testing is typically an ad hoc process where human testers manually write many test inputs and expected test results, perhaps automating their execution in a regression suite. This process is cumbersome and costly. This paper reports preliminary results on an approach to further automate this process. The approach consists of combining automated test case generation based on systematically exploring the program's input domain, with runtime analysis, where execution traces are monitored and verified against temporal logic specifications, or analyzed using advanced algorithms for detecting concurrency errors such as data races and deadlocks. The approach suggests to generate specifications dynamically per input instance rather than statically once-and-for-all. The paper describes experiments with variants of this approach in the context of two examples, a planetary rover controller and a space craft fault protection system.

1 Introduction

A program is typically tested by manually creating a *test suite*, which in turn is a set of *test cases*. An individual test case is a description of a *test input sequence* to the program, and a description of *properties* that the corresponding output is expected to have. This procedure seems complicated but ultimately unavoidable since for real systems, writing test cases is an inherently innovative process requiring human insight into the logic of the application being tested. Discussions with robotics and space craft engineers at NASA seems to support this view. However, an equally widespread opinion is that a non-trivial part of the testing work *can* be automated. In [3] is described a case study, where an 8,000 line Java application was tested by different student groups using different testing techniques. It is conjectured that the vast majority of bugs in this system that were found by the students could have been found in a fully automatic way. The paper presents reflections and preliminary work on applying low-budget

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 87–108, 2003.
© Springer-Verlag Berlin Heidelberg 2003

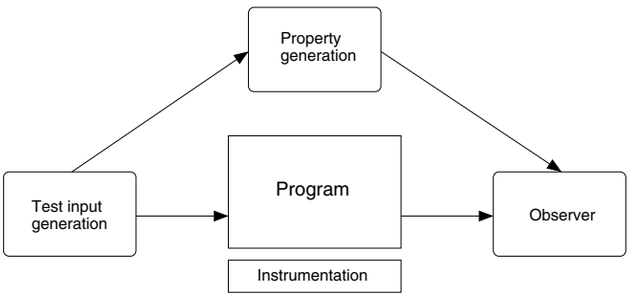


Fig. 1. Test case generation (test input generation and property generation) and runtime analysis (instrumentation and observation)

automated testing to identify bugs quickly. The paper shall be seen as a position statement of future work, based on experiments using our testing tools on case studies.

We suggest a framework for generating test cases in a fully automatic way as illustrated by Figure 1. For a particular program to be tested, one establishes a test harness consisting of four modules: a *test input generator module*, a *property generator module*, a *program instrumentation module* and an *observer module*. The test input generator automatically generates inputs to the application, one by one. A generated input is fed to the the property generator, which automatically generates a set of properties that the program should satisfy when executed on that input. The input is then fed to the program, which executes, generating an execution trace. The observer module "observes" the executing program, checking its behavior against the generated set of properties. That is, the observer takes as input an execution trace and the set of properties generated by the property generator. The program itself must be instrumented to report events that are relevant for monitoring that the properties are satisfied on a particular execution. This instrumentation can in many cases be automated. The test input generator and the property generator are both written ("hard-wired") specifically for the application that is tested. This replaces manual construction of test cases. However, the instrumentation and observer modules are generic tools that are re-used on different applications. In the rest of this paper we use the term *test case generation* to refer to test input generation and property generation, and the term *runtime analysis* to refer to instrumentation as well as observation.

The above described framework was applied to two case studies, a planetary rover controller and a space craft fault protection system. In each case the system properties were expressed in temporal logic. For the rover controller, test cases were generated using a model checker and the properties generated were *specific to a single test case*. For the fault protection system, test cases were gen-

erated by a small program, and *universal correctness properties* were manually constructed.

Property generation is the difficult step in this process. We are investigating problem characteristics and tradeoffs between the two approaches used in the studies. The approach of generating properties specific to a single test case is more novel and will be investigated further.

The paper is organized as follows. Section 2 outlines the abstract framework for test case generation that we have tried to adhere to. Section 3 describes the runtime analysis techniques. Sections 4 and 5 describe the two case studies. Finally Section 6 concludes the paper and outlines how this preliminary work will be continued.

2 Test Case Generation

This section presents, in abstract, the test case generation framework. As mentioned earlier, we consider test generation as consisting of *test input generation* and *property generation*.

2.1 Test Input Generation

In practice today, the generation of test inputs for a program under test is a time-consuming and mostly manual activity. However, test input generation naturally lends itself to automation, and therefore has been the focus of much research attention – recently it has also been adopted in industry [21, 25, 6, 10]. There are two main approaches to generating test inputs automatically: a static approach that generates inputs from some kind of model of the system (also called model-based testing), and a dynamic approach that generates tests by executing the program repeatedly, while employing criteria to rank the quality of the tests produced [20, 24]. The dynamic approach is based on the observation that test input generation can be seen as an optimization problem, where the cost function used for optimization is typically related to the code coverage (e.g. statement or branch coverage). The model-based test input (test case) generation approach is used more widely (see Hartman's survey of the field [12]). The model used for model-based testing is typically a model of expected system behavior and can be derived from a number of sources, namely, a model of the requirements, use cases, design specifications of a system [12] - even the code itself can be used to create a model (e.g. symbolic execution based approaches [19, 21]). As with the dynamic approach, it is most typical to use some notion of coverage of the model to derive test inputs, i.e., generate inputs that cover all transitions (or branches, etc.) in the model.

To construct a model of the expected system behavior can, however, be a costly process. On the other hand, generating test inputs just based on a specification of the input structure and input pre-conditions can be very effective, while typically less costly. We propose to use a model checker to traverse the space of possible valid inputs, in order to generate test inputs. We describe the

input model as a nondeterministic program that describes all valid inputs, and then we use the model checker to traverse the state space of this program. We also assert, as the property the model checker should check for, that no such test input exists – this causes the model checker to produce a counterexample whenever a valid test input has been generated and from this counterexample trace we then produce the test input. It is important that various techniques for searching the state space should be available since this gives the flexibility to generate a large array of test inputs to achieve better coverage of the behavior of the system under test. For test input generation we use the Java PathFinder model checker (JPF) that analyzes Java programs [26] and supports various heuristic search strategies (for example, based on branch coverage [11] or random search). In Section 4.2 we show how this model checker is used to generate test inputs for the Mars K9 rover.

The most closely related work to ours is the Korat tool [2] that generates test inputs from Java predicates, but instead of model checking they use a dedicated, efficient, search strategy. The use of the counterexample capability of a model checker to generate test inputs have also been studied by many others (see [18] for a good survey), but most of these are based on a full system model, not just the input structure and pre-conditions as suggested here.

2.2 Property Generation

As mentioned earlier, the ideal goal is from a particular test input to generate properties that can be checked on executions of the program on that input. More precisely, assume a particular program that we want to test and the domain *Input* of inputs. Then we have to construct the following objects. First of all, we need to define what is the domain of observable behaviors. We shall regard the executing program as generating an execution trace in the domain *Trace*, where a trace is a sequence of observable events in the domain *Event*. We must define a function:

$$\text{execute} : \mathit{Input} \rightarrow \mathit{Trace}$$

that for a given input returns the execution trace generated by the program when applied to that input. Defining the domain *Event* and the function *execute* in practice amounts to instrumenting the program to log events of importance. The resulting execution trace will then consist of these logged events. Obviously we also need to define the domain *Property* of properties that are relevant for the application and a relation: $\models \subseteq \mathit{Trace} \times \mathit{Property}$ that determines what traces satisfy what properties. We say that $(\sigma, \varphi) \in \models$, also written as $\sigma \models \varphi$, when the trace σ satisfies the property φ. Essentially what is now needed is a function translate:

$$\text{translate} : \mathit{Input} \rightarrow \mathit{Property}\textbf{\textit{-set}}$$

that for a given input returns the set of properties that it is regarded as relevant to test on the execution of the program on that input. A well-behaved program satisfies the following formula:

$$\forall i \in \mathit{Input} \cdot \forall \varphi \in \text{translate}(i) \cdot \text{execute}(i) \models \varphi$$

Our experience is that temporal logic is an appropriate notation for writing properties about the applications we have investigated, and which will be studied in subsequent sections. For a particular application one needs to provide the instrumentation (*execute*) and the property generator (*translate*), which generates a set of temporal logic properties for a particular input. We shall discuss each of these aspects in connection with the case studies.

3 Runtime Analysis

Runtime analysis consists of instrumenting the program and observing the execution of the instrumented program. The runtime analysis modules consist of a code instrumentation module, that augments the program under test with code that generates an event log, and an observer module that evaluates the event log for conformance to desired properties. The event log can be transmitted via inter-process communication or stored as a file. This allows for running an instrumented executable remotely and with little impact on the performance of the system under test.

In our case studies we used two different runtime analyzers: the commercial tool DBRover, based on an extension of the Temporal Rover tool [6] and the research tool JPaX [14, 1]. These systems will be described in the following.

The architecture of the JPaX runtime analysis framework is designed to allow several different event interpreters to be easily plugged into the observer component. In the test studies two event interpreters were used: one algorithm analyzes temporal logic properties, as already discussed, and the other concurrency properties. These algorithms are discussed below.

3.1 Instrumentation Framework

Instrumentation takes a program and the properties it is expected to satisfy and produces an instrumented version of the program that when executed generates an event log with the information required for the observer to determine the correctness of the properties. For JPaX we have implemented a very general and powerful instrumentation package for instrumenting Java bytecode.

The requirements of the instrumentation package include power, flexibility, ease of use, portability, and efficiency. We rejected alternative approaches such as instrumenting Java source code, using the debugging interface, and modifying the Java Virtual Machine because they violated one or another of these requirements.

It is essential to minimize the impact of the instrumentation on program execution. This is especially the case for real time applications, applications that may particularly benefit from this approach. Low-impact instrumentation may require careful trades between what is computed locally by the instrumentation and the amount of data that need be transmitted to the observer. The instrumentation package allows such trades to be made by allowing seamless insertion of Java code at any program point.

Code is instrumented based on an *instrument specification* that consists of a collection of predicate-action rules. The predicate is a predicate on the bytecode instructions that are the translation of a Java statement. These predicates are conjunctions of atomic predicates that include predicates that distinguish statement types, presence of method invocations, pattern-matched references to fields and local variables and so on. The actions are specifications describing the inserted instrumentation code. The actions include reporting the program point (method, and source statement number), a time stamp, the executing thread, the statement type, the value of variables or an expression, and invocation of auxiliary methods. Values of primitive types are recorded in the event log, but if the value is an object a unique integer descriptor of the object is recorded.

This has been implemented using Jtrek [5], a Java API that provides lower-level instrumentation functionality. In general, use of bytecode instrumentation, and use of Jtrek in particular, has worked out well, but there are some remaining challenges with respect to instrumenting the concurrency aspects of program execution.

3.2 Observer Framework

As described above, run time analysis is divided into two parts: instrumenting and running the instrumented program, which produces a series of events, and observing these events. The second part, event observation (see Figure 2), can be split into two stages: event analysis, which reads the events and reconstructs the run-time context, and event interpretation. Note that there may be many event interpreters.

To minimize impact on the program under test, log entries contain minimal contextual information and the log entries from different threads are interleaved. Event analysis disentangles the interleaved entries and reconstructs the context from them. The contextual information, transmitted in *internal events*, include thread names, code locations, and reentrant acquisitions of locks (lock counts). The event analysis package maintains a database with the full context of the event log. This allows for writing simpler event interpreters, which can subscribe to particular event types made accessible through an observer interface [9] and which are completely decoupled from each other.

Each event interpreter builds its own model of the event trace, which may consist of dependency graphs or other data structures. It is up to the event interpreter to record all relevant information for keeping a history of the events, since the context maintained by the event analysis changes dynamically with the event evaluation. Any information that needs to be kept for the final output, in addition to the context information, needs to be stored by the event interpreter. If an analysis detects violations of its rules in the model, it can then show the results using the stored data and context information such as code locations, thread names, etc.

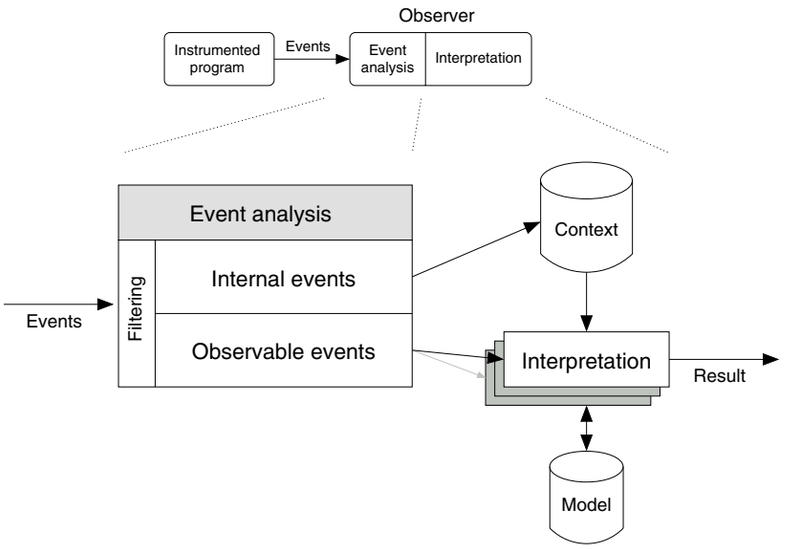


Fig. 2. The observer architecture

3.3 Temporal Logic Monitoring

Temporal logic in general, and Linear-time Temporal Logic (LTL) in particular, has been investigated for the last twenty years as a specification language for reactive systems [22]. LTL is a propositional logic with the standard connectives $\wedge$, $\vee$, $\rightarrow$ and $\neg$. It furthermore includes four temporal operators: $\Box p$ (always p), $\Diamond p$ (eventually p), $p\ \mathcal{U}\ q$ (p until q – and q has to eventually occur), $\circ p$ (in next step p), and four dual past-time operators (always p in the past, p some time in the past, p since q, and in previous step p). As an example, consider the future time formula $\Box(p \rightarrow \Diamond q)$. It states that it is always the case ($\Box$), that when p holds, then eventually ($\Diamond$) q holds. LTL has the property of being intuitively similar to natural language and capable of describing many interesting properties of reactive systems.

Metric Temporal Logic (MTL) extends LTL so that every temporal operator can be augmented with a relative-time or real-time constraint. Hence, for example, $p\,\mathcal{U}_{c<5}\,q$ means p must be true until a future time, at most 5 c real-time units in the future, where q must hold. Here c is some clock. Similarly $p\,\mathcal{U}_{<5}\,q$ requires q to be true at most 5 cycles in the future, using the underlying state, or cycle based semantics to define the notion of a cycle. As mentioned, we have experimented with two systems that perform temporal logic monitoring: DBRover [6] and JPaX [14, 1].

The DBRover Temporal Observer The DBRover is a MTL monitoring tool, based on the TemporalRover code generator [6]. DBRover extends MTL with two forms of parametrization: multi-instancing, which allows for a rule to be independently validated per instance of an object class, process, or thread, and parametrization based on time series data values [7], which enables the verification of properties such as temporal stability and monotonicity. It consists of a GUI for editing temporal assertions, a graphical LTL/MTL simulator, and an execution engine. The DBRover builds and executes temporal rules for a target program or application; at run time, the DBRover listens for event messages from the target application and evaluates corresponding temporal assertions.

The JPaX Temporal Observer With respect to temporal logics, we have implemented several specialized algorithms in JPaX: traversing the execution trace either forwards or backwards, based on either rewriting or automata generation, implemented in either Java or Maude [4]. We next briefly sketch one of these algorithms and refer the interested readers to more elaborated descriptions. Efficiency of runtime analysis algorithms is always an important aspect of our research, even if the observer operates off-line. A crucial observation is that one can design more efficient algorithms if one focusses on segments of temporal logics rather than on the entire logic. Thus, we were able to develop optimal algorithms for future time and for past time temporal logics separately. We do not regard this segmentation as a problem in practice, because in our experience so far one rarely or never uses both future and past time operators in the same requirements formula. The algorithm we are going to describe monitors future time temporal logic formulas and is entirely based on rewriting technology. The idea is to maintain a set of monitoring requirements as future time LTL formulas and modify them accordingly when a new event is emitted by the instrumented program. If one of these formulas ever becomes *false* then it means that that formula has been violated, so an error message is generated and an appropriate action is taken. Four rewriting rules, inspired from known recurrences of temporal operators, transform the formulas whenever a new nonterminal event e is received (and four others not mention here are called on terminal events):

$$\begin{array}{ll}
(\circ F)\{e\} & \rightarrow F, \\
(\Box F)\{e\} & \rightarrow F\{e\} \wedge \Box F, \\
(\Diamond F)\{e\} & \rightarrow F\{e\} \vee \Diamond F, \\
(F\ \mathcal{U}\ F')\{e\} & \rightarrow F'\{e\} \vee (F\{e\} \wedge F\ \mathcal{U}\ F')
\end{array}$$

The formula $F\{e\}$, for some formula F, is the (transformed) formula which should hold next, after receiving the event e. For example, for $\Diamond F$ to hold now, where the first event in the remaining trace is e, either F must hold now ($F\{e\}$), or $\Diamond F$ must again hold in the future, thus postponing the obligation. Using memoization (or hashing) techniques provided by advanced rewriting engines such as Maude, the simple rewriting algorithm above performs well in practice. We were for example able to monitor 100 million events in less than 3 minutes on a formula $\Box(g \rightarrow (\neg r)\ \mathcal{U}\ y)$ stating a safety policy of a traffic light controller

(yellow should come after green). The interested reader is referred to [15, 16] for proofs of correctness, complexity analysis and evaluation of this algorithm.

A second approach to building LTL observers based on automata construction is found in [16]. A rewriting-based algorithm for monitoring past time LTL requirements formulas has been presented in [13], which is quite different from the one for future time LTL. A dynamic algorithm approach to monitoring past time LTL formulas is presented in [17].

3.4 The JPaX Concurrency Analyzer

Multi-threaded programs are particularly difficult to test due to the fact that they are non-deterministic. A multi-threaded program consists of several threads that execute in parallel. A main issue for a programmer of a multi-threaded application is to ensure mutual exclusion to shared objects. That is: to avoid data races where one thread writes to an object while other threads simultaneously either write to or read the same object. Multi-threading programming languages therefore provide constructs for ensuring mutual exclusion. To ensure mutual exclusion on a shared object, a thread can acquire a lock before accessing the object, releasing it after. If other threads acquire the same lock when accessing the object, mutual exclusion is guaranteed. If threads do not acquire the same lock (or don't acquire locks at all) when accessing an object then there is a risk of a data race. The Eraser algorithm [23] can detect such disagreements by analyzing single execution traces. The Eraser algorithm has been implemented in the JPaX tool.

Deadlocks can occur when two or more threads acquire locks in a cyclic manner. As an example of such a situation consider two threads T_1 and T_2 both acquiring locks A and B. Thread T_1 acquires first A and then B before releasing A. Thread T_2 acquires B and then A before releasing B. This situation poses a deadlock situation since thread T_1 can acquire A whereafter thread T_2 acquires B, where after both threads cannot progress further. In JPaX we have implemented an algorithm for detecting such deadlock situations. It builds a lock graph, where nodes are locks and edges represent the lock hierarchy. That is, for the above example, there will be an edge from A to B and another edge from B to A. Hence for this example the graph contains a cycle, and a cycle represents a potential deadlock situation. This algorithm yields false positives (false warnings) and false negatives (missed deadlocks). In [1] an extension to this algorithm is described that reduces the number of false positives. JPaX's concurrency analysis has been integrated with the DBRover, where deadlock results are graphically displayed as UML message sequence charts.

4 Case Study 1: A Planetary Rover Controller

Our first case study is the planetary rover controller K9, and in particular its executive subsystem, developed at NASA Ames Research Center – a full account of this case study is described in [3]. The executive receives plans of actions that

Plan → *Node*

Node → *Block* | *Task*

Block → `(block`
NodeAttr
`:node-list (`*NodeList*`))`

NodeList → *Node NodeList* | ϵ

Task → `(task`
NodeAttr
`:action` *Symbol*
[`:fail`]
[`:duration` *DurationTime*`]`)

NodeAttr → `:id` *Symbol*
[`:start-condition` *Condition*]
[`:end-condition` *Condition*]
[`:continue-on-failure`]

Condition → `(time` *StartTime EndTime*`)`

```
(block
 :id plan
 :continue-on-failure
 :node-list (
   (task
    :id drive1
    :start-condition (time +1 +5)
    :end-condition  (time +1 +30)
    :action BaseMove1
    :duration 20
   )
   (task
    :id drive2
    :end-condition (time +10 +16)
    :action BaseMove2
    :fail
   )
  )
)
```

Fig. 3. Plan grammar (left) and an example of a plan (right)

the rover is requested to carry out, and executes these plans. First we present a description of the system, including a description of what plans (the input domain) look like. Then we outline how plans (test inputs) can be automatically generated using model checking, and finally we describe how for each plan a set of temporal logic properties can be automatically generated, that the executive must satisfy when executing the plan.

4.1 System Description

The executive is a multi-threaded system (8,000 lines of Java code) that receives flexible plans from a planner, which it executes according to a plan language semantics. A plan is a hierarchical structure of actions that the rover must perform. Traditionally, plans are deterministic sequences of actions. However, increased rover autonomy requires added flexibility. The plan language therefore allows for branching based on conditions that need to be checked, and also for flexibility with respect to the starting time and ending time of an action. We give here a short presentation of the (simplified) language used in the description of the plans that the rover executive must execute.

Plan Syntax A plan is a *node*; a node is either a *task*, corresponding to an *action* to be executed, or a *block*, corresponding to a logical group of nodes.

Figure 3 (left) shows the grammar for the language; we should note that all the node attributes, with the exception of the node's *id*, are optional. Each node may specify a set of *conditions*, e.g., the *start condition* (that must be true at the beginning of the node execution) and the *end condition* (that must be true at the end of the node execution). Each condition includes information about a relative or absolute time window, indicating a lower and an upper bound on the time. The *continue-on-failure* flag indicates what the behavior will be when node failure is encountered.

The attributes *fail* and *duration* were added to the original plan syntax to facilitate testing of the executive. That is, during testing using test case generation, the real actions are never executed since this would require operating the rover mechanics. The `:fail` and `:duration` attributes replace the actions during testing. The *fail* flag for a task specifies the action status after execution; the *duration* specifies the duration of the action. Figure 3 (right) shows a plan that has one block with two tasks (`drive1` and `drive2`). The time windows here are relative (indicated by the '`+`' signs in the conditions).

Plan Semantics For every node, execution proceeds through the following steps: (1) Wait until the start condition is satisfied; if the current time passes the end of the start condition, the node times out and this is a node failure. (2) The execution of a *task* proceeds by invoking the corresponding action. The action takes exactly the time specified in the `:duration` attribute. Note that this attribute during testing replaces the actual execution of the action on the rover. The action's status must be fail, if `:fail` is true or the time conditions are not met; otherwise, the status must be success. If the action's status indicates failure, we have a task failure. The execution of a *block* simply proceeds by executing each of the nodes in the node-list in order. (3) If the time exceeds the end condition, the node fails.

On a *node failure*, when execution returns to the sequence, the value of the failed node's *continue-on-failure* flag is checked. If true, execution proceeds to the next node in the sequence. Otherwise the node failure is propagated to any enclosing nodes. If the node failure passes out to the top level of the plan, the remainder of the plan is aborted.

4.2 Test Input Generation

Figure 4 shows the Java code, referred to as the *universal planner*, that we used to generate plans (i.e., test inputs for the executive). We exploit nondeterminism (i.e., `choose` methods) to systematically generate all the possible plans (up to a given number of nodes specified by `nNodes`) that have the structure specified by the grammar in Figure 3; we also use nondeterminism to generate the data values for the time conditions (up to a given value specified by `tRange`). The assertion in the program specifies that it is not possible to create a "valid" plan (i.e., executions that reach this assertion generate valid plans). We used the JPF model checker to explore the (finite) state space of the generated input plans (i.e.,

```
class UniversalPlanner { ...
  static int nNodes = 0;
  static int tRange = 0;

  static void Plan(int nn, int tr) {
    nNodes = nn; tRange = tr;
    Node plan = UniversalNode();
    print(plan);
    assert(false);
  }

  static Node UniversalNode() {
    if (nNodes == 0) return null;
    if (chooseBool()) return null;
    if (chooseBool())
      return UniversalTask();
    return UniversalBlock();
  }

  static Node UniversalTask() {
    UniversalAttributes();
    Symbol action = new Symbol();
    boolean fail = chooseBool();
    int duration = choose(tRange);
    Task t =
      new Task(id, action, start,
               end, continueOnFailure,
               fail, duration);
    nNodes--;
    return t;
  }

  static Node UniversalBlock(){
    UniversalAttributes();
    nNodes--;
    ListOfNodes l = new ListOfNodes();
    for (Node n = UniversalNode();
         n != null; n = UniversalNode())
      l.pushEnd(n);
    Block b = new Block(id, l, start, end,
                        continueOnFailure);
    return b;
  }

  static Symbol id;
  static TimeCondition start, end;
  static boolean continueOnFailure;

  static UniversalAttributes() {
    id = new Symbol();
    int time1 = choose(tRange);
    int time2 = time1 + choose(tRange);
    start = new TimeCondition(time1,time2);
    time1 = choose(tRange);
    time2 = time1 + choose(tRange);
    end = new TimeCondition(time1,time2);
    continueOnFailure = chooseBool();
  }
}
```

Fig. 4. Universal planner that generates input plans for system under test

JPF model checks the universal planner). We used different search strategies to find multiple counterexamples (to the assertion); for each counterexample we ran JPF in simulation mode to `print` the generated plans to a file, which then served as input to the rover.

4.3 System Analysis

The semantics of a particular plan can very naturally be formulated in temporal logic. In writing such properties, we used the following predicates: start(id) (true immediately after the start of the execution of the node with the corresponding id), success(id) (true when the execution of the node ends successfully), fail(id) (true when the execution of the node ends with a failure); end(id) denotes success(id) $\vee$ fail(id). We instrumented the code to monitor these predicates. For each plan we further wrote a collection of temporal properties over these predicates and verified their validity on execution traces. As an example, the properties for the plan shown in Figure 3 (right) are shown in Figure 5.

- $\Diamond$start(*plan*), i.e., the initial node `plan` should eventually start.
- $\Box$(start(*plan*) $\rightarrow \Diamond_{1,5}$start(*drive1*)), i.e., if the `plan` starts, then task `drive1` should begin execution within 1 and 5 time units.
- $\Box$(start(*drive1*) $\rightarrow$ ($\Diamond_{1,30}$success(*drive1*) $\vee \Diamond$fail(*drive1*))), i.e., if task `drive1` starts, then it should end successfully within 1 and 30 time units or it should eventually terminate with a failure.
- $\Box$(success(*drive1*) $\rightarrow \Diamond$start(*drive2*)), i.e., if task `drive1` ends successfully, then task `drive2` should eventually begin execution.
- $\Box$(end(*drive2*) $\rightarrow \Diamond$success(*plan*)), i.e., termination of task `drive2` implies successful termination of the whole plan (due to `continue-on-failure` flag).
- $\Diamond$success(*drive1*), i.e., task `drive1` should end successfully (since `:duration` is within time window).
- $\Diamond$fail(*drive2*), i.e., task `drive2` should fail (due to `:fail`).

Fig. 5. Temporal logic properties representing partial semantics of plan in Figure 3

This set of properties does not fully represent the semantics of the plan, but the approach appeared to be sufficient to catch a large amount of bugs.
The purpose of the case study was to find a number of seeded errors in the rover by using a number of different technologies, including runtime analysis, model checking, static analysis and traditional testing. Here we just focus on the results on the runtime analysis which was done according to the framework described in this paper. We used the DBRover to monitor the temporal properties that each plan had to satisfy. We generated the formulas for each plan by hand, in a similar fashion as the formulas given in Figure 5. Using this approach most of the errors pertaining to plan semantics were easily discovered. It is worth noting that none of the other techniques found as many of the plan semantic errors as the temporal logic runtime analysis technique. It is possible to generate these properties automatically.

The rest of the errors were related to concurrency and for those JPaX's concurrency analysis was used (see Section 3.4). All but two of the concurrency errors were discovered this way. JPaX found all the seeded data races and all the seeded cyclic deadlocks (resource deadlocks), which are the errors this tool can find. The errors it could not find were communication deadlocks involving Java's **wait** and **notify** constructs. These errors were very subtle and required analysis beyond what JPaX could do at the time. Note that this supports our hypothesis that our framework is good at finding a large class of errors, but may not be as good in finding some subtle errors – for this more advanced techniques may be required, such as model checking (that did find all the concurrency errors in the rover).

5 Case Study 2: A Space Craft Fault Protection System

5.1 System Description

In this section we describe the ongoing application of these ideas to a verification experiment targeted to mission-critical NASA application called a fault protection system (FPS). A fault protection system monitors critical hardware and software components of a spacecraft and executes corrective responses to detected faults.

The FPS system we analyze consists of a reusable core engine and portion customizable for a specific spacecraft. The FPS is a complex, logic-oriented system. That is, the code is dense with decision points yielding a very large number of potential control paths. Given the complexity, mission criticality, and reusability of this code, a high level of reliability is demanded. The FPS system has been the study of a model checking experiment reported in [**?**].

The FPS was originally written in C and has flown on space missions. A new implementation in Java has recently been developed. This implementation was provided with a wrapper that allows it to run stand alone using a simple command interface. We based our experiments on this Java implementation.

One of the commands informs the FPS of problem symptom. The FPS maintains a customizable mapping between symptoms, faults, and responses. The basic mode of execution is to respond to a symptom by maping the symptom to a fault, the fault to a response and then execute the response. The FPS executes one response at a time. However it may have reported to many symptoms that require multiple responses. Thus responses must be queued. In addition, responses have different priority levels; some responses can be interrupted; and some responses "call" other responses as subroutines. Responses, symptoms, and faults, may be in various modes, such as enabled or disabled, and there are commands to reset responses and the whole fault protection system. Furthermore, the FPS must respond to ground commands to execute a response in a manner that differs from responses invoked by fault protection itself. It is these capabilities and others that contribute to the significant complexity of the FPS.

The FPS core engine is specified as a number of finite state machines that signal each other by generating events. There are state machines for the FPS engine itself, instances of a state machine for each symptom, fault and response. In fact there are two state machines for each response. One is part of the engine itself (referred to in the previous sentence) — it records the status of the response, and a second encodes the semantics of the responses themselves (e.g. turn on a device). Code is automatically generated for the latter response state machines and become part of the code base for an instantiation of the FPS. In fact we analyzed a particular instantiation of FPS with two responses.

Our approach to this experiment is to

1. write a small program to generate test cases for the FPS,
2. write a partial specification (of an instantiation) of the FPS in the form of LTL (linear temporal logic) assertions,

3. instrument the FPS to emit a log that allows checking of the LTL assertions,
4. write a test driver which runs each generated test, and invokes the LTL observer to analyze the generated log,
5. and finally, the observer reports those test cases and assertion combinations that failed or are inconclusive.

This process will be explicated in the subsequent sections.

5.2 Test Case Generation

For this application a simple program (less than 200 lines of Python code) was written to generate test cases. A test case for the FPS is simply a sequence of commands. The generator generated commands sequences of a specified length by making independent random selections of individual commands. The generator was parameterized by

- the number of test cases to be generated,
- the number of commands in a test case,
- the relative frequency of each command,
- for each command, the relative frequency of its arguments.

There was one subtlety to this process. It should be noted that the execution of a response takes time. Generally in one time unit a response will perform the action described in a single state and then transition to a new state. The signalling of a new time step corresponds to the execution of a command (or signal) to advance the clock. Thus the relative frequencies of the commands needs to be selected such that response queues do not overflow, yet there would be enough pending responses to test the behavior when multiple faults required a response.

We thank Owen O'Mally of NASA for writing the test case generator.

5.3 Code Instrumentation

The FPS system is instrumented such that as it executes, it writes out a log with sufficient information to allow run time analysis to determine whether the program execution satisfies the LTL assertions. Since the new instrumentation package described above, has not yet been integrated with the old version of JPaX that we are using, we used an older instrumentation package. A significant restriction of this package is that it limits program variables that may be observed to those of integer type. Thus the FPS code had to be extended to represent conditions and events referenced in the LTL asertions as newly introduced integer variables.

5.4 Runtime Analysis

We used the JPaX runtime analyzer to check LTL formulas against the execution traces generated for each test execution.

JPaX is a system developed by the ASE group at NASA Ames. coordiate with arlier sections....

maude LTL specification ....

LTL specification In this experiment the test oracle is a partial LTL specfication for the FPS. As such it is not specific to any single input as is the case for the K9 experiment. The specification was written after the fact, that is after the software itself for the purpose of this experiment. It was based on the code itself, design documents describing the state machine descriptions for the FPS, and some test data and expected results. Clearly, the purpose of the specification is to describe the intended semantics of the program independently of the program itself, and so using the program is less than ideal. However, we found in places that the code itself exhibits inconsistencies. Resolving these was essential to specification creation. In fact it was by this process that some bugs were found.

In our contact with the developers of the Java version of the FPS code we encouraged them to write such LTL assertions but found this difficult. However, LTL is a very natural notation for describing FPS. FPS is a transaction system. Its top level functionality is as a command processor. It reads commands and executes them, sometimes by queing an action. Characterizing the state of a FPS (eg as the state of each state machine and the value of certain variables) and then characterizing the action over time of each command on the state is a natural methdology for developing such a specification. One subtlety of the process is the mapping of specification level concepts (eg a response completion) with a suitable code-level event. There may be many candidate predicates or events at the code level that correspond to a specification level predicate or event. These code level predicates may differ in subtle ways and may correctly model intended semantics. We found it most effective to relate predicates and events to the inputs and outputs of the computation when possible.

Putting it all together A simple make file was used to instrument the code, run and analyze each of the test cases in turn and report the results.

5.5 Results

These experiments are still in progress. All of the components described are in place and executing. The additional work is to strengthen the LTL specification. However, to date we have already identified program bugs and inaccuracies in the specifications. One bug is significant. It is not one introduced in the Java version but existed in the C version that flew on spacecraft.

This represents a large-scale application of tools we developed. We were aware of some of the tool limitations, and others became apparent:

- The tools need to be easier to use. It should not be necessary to modify the code of the system under test.
- A clearer distinction between events and conditions must be explained.
- There must be a clear way of describing that only "required" actions are taken, and that all others are not. I.e., the software does only what is intended to do and nothing more.

In this section we describe the ongoing application of the framework in a verification experiment targeted to a mission-critical NASA application called a fault protection system (FPS). A fault protection system monitors critical hardware and software components of a spacecraft and executes corrective responses to detected faults. Our approach to this case study is to:

1. write a small program to generate test cases for the FPS,
2. write a partial specification (of an instantiation) of the FPS in the form of LTL (linear temporal logic) assertions,
3. instrument the FPS to emit a log that allows checking of the LTL assertions,
4. write a test driver which runs each generated test, and invokes the LTL observer to analyze the generated log,
5. and finally, the observer reports those test cases and assertion combinations that failed or are inconclusive.

5.6 System Description

The FPS system we analyze consists of a reusable core engine and a portion customizable for a specific spacecraft. The FPS is a complex, logic-oriented system. That is, the code is dense with decision points yielding a very large number of potential control paths. Given the complexity, mission criticality, and reusability of this code, a high level of reliability is demanded. The FPS system has been the study of a model checking experiment reported in [8]. The FPS was originally written in C and has flown on spacecraft. A new implementation in Java has recently been developed. This implementation was provided with a wrapper that allows it to run stand alone using a simple command interface. We based our experiments on this Java implementation.

One of the commands informs the FPS of spacecraft problem symptoms. The FPS maintains a customizable mapping between symptoms, faults, and responses. The basic mode of execution is to respond to a symptom by mapping the symptom to a fault, the fault to a response and then execute the response. The FPS executes one response at a time. However many symptoms may have been reported requiring that responses be queued. In addition, responses have different priority levels; some responses can be interrupted; and some responses "call" other responses as subroutines. In addition, responses, symptoms, and faults, may be in various modes, such as enabled or disabled, and there are commands to reset responses and the whole fault protection system. Furthermore, the FPS must respond to ground commands to execute a response in a manner that differs from responses invoked by fault protection itself. It is these capabilities and others that contribute to the significant complexity of the FPS.

The FPS core engine is specified as finite state machines that signal and invoke each other by generating events. There are state machines for the FPS engine itself, instances of a state machine for each symptom, fault and response. In fact there are two state machines for each response. One is part of the engine itself – it records the status of the response, and a second encodes the semantics of the responses themselves (e.g. turn on a device). Code is automatically generated for the latter response state machines and become part of the code base for an instantiation of the FPS. In fact we analyzed a particular instantiation of FPS with two responses.

5.7 Test Input Generation

For this application a simple program (less than 200 lines of Python code) was written to generate test cases. A test case for the FPS is simply a sequence of commands. The generator generated commands sequences of a specified length by making independent random selections of individual commands. The generator is parameterized by:

- the number of test cases to be generated,
- the number of commands in a test case,
- the relative frequency of each command,
- for each command, the relative frequency of its arguments.

There is subtlety to this process. It should be noted that the execution of a response takes time. Generally in one time unit a response performs the action described in a single state of its state machine and then transitions to a new state. The signalling of a new time step corresponds to the execution of a command (or signal) to advance the clock. Thus the relative frequencies of the commands need to be selected such that response queues do not overflow, yet there would be enough pending responses to test the behavior when multiple faults required a response. We thank Owen O'Malley of NASA for writing the test case generator.

5.8 System Analysis

We used the JPaX temporal observer to check LTL formulas against the execution traces generated for each test execution. We used a partial static LTL specification for the FPS. As such it is not specific to any single input as is the case for the K9 experiment. It is based on the code itself, design documents describing the state machines of the FPS, and some test data and expected results. Clearly, the purpose of the specification is to describe the intended semantics of the program independently of the program itself, and so using the program is less than ideal. However we found in places that the code itself exhibits inconsistencies and resolving these was essential to specification creation. In fact, it was by this process that some bugs were found.

LTL is a very natural notation for describing the FPS. The FPS is a transaction system. Its top level functionality is as a command processor. It reads

commands and executes them, potentially queuing actions. Characterizing the state of the FPS (e.g., as the state of each state machine and the value of certain variables) and then characterizing the action over time of each command on the state is a natural methodology for developing such a specification. In addition, properties asserting that each action taken is in response to a relevant command will insure that the program does only what is intended to do and nothing more. One subtlety of the process is the mapping of specification level concepts (e.g. a "response completes") with a suitable code-level event. There may be many candidate predicates or events at the code level that correspond to a specification level predicate or event. These code level predicates may differ in subtle ways and may or may not correctly model intended semantics. We found it most effective to relate predicates and events closely to the inputs and output products of the program.

These experiments are still in progress. All of the components described are in place and executing. The additional work is to strengthen the LTL specification. However, to date we have already identified program bugs and inaccuracies in the specifications. One bug is significant in that it existed in the C version that flew on spacecraft.

This case study varies from the proposed framework in two respects. First we did not use a model checker to generate test cases. The model checking approach would work fine, and would enjoy the benefits described above. Second the defined properties are universal correctness properties, valid for all inputs. The framework suggests defining a function that maps an input to a set of properties specific to that input. In this case it was straightforward to define universal properties and it is not evident that the alternative approach makes property specification any easier. Note that the specification was constructed by formalizing the semantics of each type of command, taking into account the computation state, independently of the other commands. If this approach is followed then knowing the other commands in the sequence would not be of much help in simplifying the property set.

6 Conclusions and Future Work

We have presented a framework for testing based on automated test case generation and runtime analysis. The framework requires construction of a test input generator and a property generator for the application. From that, an arbitrarily large test suite can be automatically generated, executed and verified to be in conformity with the properties. For each input sequence (generated by the test input generator) the property generator constructs a set of properties that must hold when the program under test is executed on that input. The program is instrumented to emit an execution log of events. An observer checks that the event log satisfies the set of properties.

We take the position that writing test oracles as temporal logic formulas is both natural and leverages algorithms that efficiently check if execution on a test input conforms to the properties. While property definition is often difficult, at

least for some domains, an effective approach is to write a property generator, rather than a universal set of properties that are independent of the test input. Note also that the properties need not completely characterize correct execution. Instead, a user can choose among a spectrum of weak but easily generated properties to strong properties that may require construction of complex formulas. We see the proposed framework as a complementary approach to testing that may be applied opportunistically to test selected system behaviors. We proposed and demonstrated the use of model checking for test case generation. We will be exploring how to improve the quality of the generated test suite by altering the model checker's search strategy, and by use of symbolic execution.

In the near future, we will continue the development of a complete testing environment for the K9 rover and seek to get this technology transferred to NASA engineers. We are continuing the work on instrumentation of Java bytecode. In a parallel effort with Robert Filman, one of our NASA Ames (RIACS) colleagues, a source code instrumentation tool is being developed that is based on ideas in Aspect Oriented Programming. Concerning the observer part, we have presented various interpreters for temporal logic. Current and future work is devoted to the design of a specification language that is attractive to engineers and powerful enough to capture a majority of practical system requirements. We are also investigating new algorithms for concurrency analysis that extend the scope of deadlocks and data races that can be identified.

Our research group has done fundamental research in other areas, such as software model checking (model checking the application itself, and not just the input domain), and static analysis. In general, our ultimate goal is to combine the different technologies into a single coherent framework.

References

[1] S. Bensalem and K. Havelund. Reducing False Positives in Runtime Analysis of Deadlocks. Submitted for publication, October 2002. 91, 93, 95

[2] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002. 90

[3] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M.Lowry, C. Pasareanu, A. Venet, and W. Visser. A Comparative Field Study of Advanced Verification Technologies. Internal report, in preparation for submission, November 2002. 87, 95

[4] M. Clavel, F. J. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and Programming in Rewriting Logic, March 1999. Maude System documentation at `maude.csl.sri.com/papers`. 94

[5] S. Cohen. Jtrek. Compaq, `www.compaq.com/java/download/jtrek`. 92

[6] D. Drusinsky. The Temporal Rover and the ATG Rover. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*, pages 323–330. Springer, 2000. 89, 91, 93, 94

[7] D. Drusinsky. Monitoring Temporal Rules with Temporal Data. Submitted for publication, October 2002. 94

[8] M. Feather, S. Fickas, and N. Razermera-Mamy. Model-Checking for Validation of a Fault Protection System. In *Proceedings of Sixth IEEE International Symposium on High Assurance System Engineering*. IEEE Computer Society, October 2001. 103

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. 92

[10] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, July 2002. 89

[11] A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, July 2002. 90

[12] A. Hartman. Model Based Test Generation Tools. `www.agedis.de/documents/ModelBasedTestGenerationTools`$_c$ *s.pdf* + . 89

[13] K. Havelund, S. Johnson, and G. Roşu. Specification and Error Pattern Based Program Monitoring. In *Proceedings of the European Space Agency workshop on On-Board Autonomy*, Noordwijk, The Netherlands, October 2001. 95

[14] K. Havelund and G. Roşu. Monitoring Java Programs with Java PathExplorer. In K. Havelund and G. Roşu, editors, *Proceedings of the First International Workshop on Runtime Verification (RV'01)*, volume 55 of *Electronic Notes in Theoretical Computer Science*, pages 97–114, Paris, France, July 2001. Elsevier Science. 91, 93

[15] K. Havelund and G. Roşu. Monitoring Programs using Rewriting. In *Proceedings of the International Conference on Automated Software Engineering (ASE'01)*, pages 135–143. IEEE CS Press, 2001. Coronado Island, California. 95

[16] K. Havelund and G. Roşu. A Rewriting-based Approach to Trace Analysis. Submitted for journal publication, September 2002. 95

[17] K. Havelund and G. Roşu. Synthesizing Monitors for Safety Properties. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280 of *Lecture Notes in Computer Science*, pages 342–356. Springer, 2002. EASST best paper award at ETAPS'02. 95

[18] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A Temporal Logic Based Theory of Test Coverage and Generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, April 2002. 90

[19] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976. 89

[20] B. Korel. Automated Software Test Data Generation. *IEEE Transaction on Software Engineering*, 16(8):870–879, August 1990. 89

[21] Parasoft. `www.parasoft.com`. 89

[22] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77, 1977. 93

[23] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997. 95

[24] N. Tracey, J. Clark, and K. Mander. The Way Forward for Unifying Dynamic Test-Case Generation: The Optimisation-Based Approach. In *International Workshop on Dependable Computing and Its Applications (DCIA)*, pages 169–180. IFIP, January 1998. 89

[25] T-VEC. `www.t-vec.com`. 89

[26] W. Visser, K. Havelund, G. Brat, and S. Park. Model Checking Programs. In *Proceedings of ASE'2000: The 15th IEEE International Conference on Automated Software Engineering*. IEEE CS Press, September 2000. 90

A Framework for Proving Contract-Equipped Classes

Bertrand Meyer
ETH Zürich, Chair of Software Engineering
http://se.inf.ethz.ch
(also Eiffel Software, Santa Barbara, and Monash University)

Abstract. As part of a general effort to provide a new basis for software development through reuse of "Trusted Components", we outline a scheme for proving that classes equipped with contracts in the Eiffel style meet these contracts. The approach takes advantage of the inheritance structure to separate proof obligations between deferred (abstract) classes, to be validated against a model, and their effective implementations, which then must only be proved against the contracts of the deferred ancestors. The testbed for this study is the EiffelBase library of fundamental data structures and algorithms, whose classes include extensive contracts.

1 Trusted Components, Low Road and High Road

The techniques proposed here are part of a general effort to develop *Trusted Components* [8]: reusable software elements with guaranteed properties. We start with an overview of this broader goal and of the present work's place in it.

Work on Trusted Components rests on the observation that one of the most realistic hopes for radical improvements in software quality and productivity is to combine *reuse* and *quality*. Reuse ensures faster time to completion, economies of scale, and the opportunity to turn expertise into concrete assets. But reuse scales up everything, deficiencies included; reusable components should be subjected to quality criteria far more rigorous than ordinary non-reusable software. The effect of scale then becomes a benefit rather than a risk: every reusing application profits from the quality investment made in the reusable components.

There has been widespread advocacy for the idea of reuse, especially of object-oriented classes [3] and binary components [13]. A real market for components has emerged in recent years, although the need to associate quality with reuse doesn't seem to have registered with the software industry at large. Trusted Component efforts attempt to remedy this mistake and to provide a solid basis of high-quality components for software development.

Ideally, a Trusted Component should have a formal specification of its relevant properties and then a proof that its implementation satisfies the specification. This is indeed the approach explored in the following sections. It assumes, however, components that are designed from the outset with this goal in mind; while important for the long term, this view does little to address the concerns of industry developers who have to work with existing components — object-oriented, EJB, COM, .NET … — that have not been built accordingly; if nothing else, most commercial components do not publish their source code, making third-party proofs impossible.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 108-125, 2003.
© Springer-Verlag Berlin Heidelberg 2003

As a result we face a choice between providing full proofs of components produced specifically for this purpose ("*software as it should be*") and attempting partial qualification of existing commercial components built with far more informal techniques ("*software as it is*"). Our work on Trusted Components recognizes the need for both of these approaches, and as a consequence pursues two tracks at once:

- A *high road* for producing components whose correctness can be proved, and indeed proving it.
- A *low road* for assessing properties of existing components, commercial or open-source. It will generally be impossible to include correctness proofs in this case, but we can still define relevant quality criteria — extent of functional and performance specification, examples of use, documentation, evidence of prior reuse, extendibility … — and assess components against them. A *Component Quality Model* is under development for this purpose [10].

The work presented here is on the "high road". Focused on fundamental components covering common data structures and algorithmes, it attempts to produce full proofs of their correctness.

An important practical issue of correctness proofs for object-oriented components is the matter of run-time structures, which typically involves extensive use of *pointers* (or "references"). In parallel with the work described here, a theory has been developed to model pointer-rich object structures and prove the corresponding software properties. Described in a separate paper [11], it is closely connected to the present work, and plays an important part in its application to actual proofs of object-oriented software.

2 Formal Methods for Reusable Components

Most applications of formal methods so far have been to systems or subsystems, generally in mission-critical areas where the investment in formality and proofs is justified by the grave consequences that malfunctions could cause. Examples include defense applications, transportation systems such as those built with the B approach [1], and Java Bytecode verification through Abstract State Machines [12]. With components, the economic justification is simply the effect of scale arising from reuse. A reusable component may not be by itself "critical" in the same sense as the system controlling the closing of doors in a train, but the effect of a deficiency could be just as bad given the number and scope of applications that may rely on it.

Proof technology has now advanced to a point where it appear practical enough to go beyond these applications and consider proofs of components. Although not extensively explored until now, this combination of formal methods and reuse offers one of the most interesting outlets for formal development.

In pursuing it we will rely on Eiffel's **Design by Contract**™ [4] [5] [7]. Classes equipped with contracts — routine preconditions, routine postconditions and class invariants — already possess a degree of formal specification.

Although devised from their origin with the ultimate goal of permitting proofs, contracts have been used so far in Eiffel for other purposes [7]: as a design method to obtain correct software; as a documentation technique in connection with the tools of the EiffelStudio environment; as a central part of the testing, debugging and quality assurance process; as a management aid; as a guide for using inheritance properly; and as a basis for exception handling relying on a clear definition of what constitutes a "normal" and "abnormal" case.

3 Layers of Proof

To prove properties of components (and in fact of applications too) it appears desirable, because of the potential complexity of some libraries, to organize the proof process in a hierarchical manner, where the basic (lowermost) levels are very close to mathematical concepts, and each higher level is proved conditional to the correctness of the lower one:

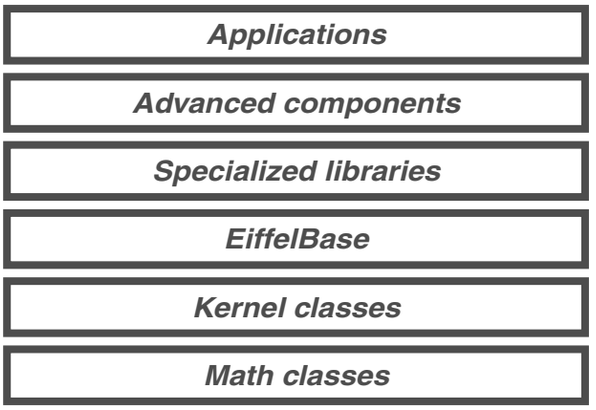


Figure 1: Application and component layers

4 EiffelBase

Of particular interest among the layers of figure 1 is the EiffelBase library, the focus of our first investigation. The library results from a systematic approach, detailed in reference [6], at providing a "Linnaean" taxonomy of the fundamental structures of computing science. EiffelBase takes full advantage of object-oriented mechanisms and is thus a good testbed for the scalability of any techniques developed. Figure 2 describes the hierarchy of its top inheritance levels.

EiffelBase classes are extensively equipped with contracts, as illustrated by the examples reviewed below. This makes them a prime target for proofs, since the properties to prove are already part of the class text.

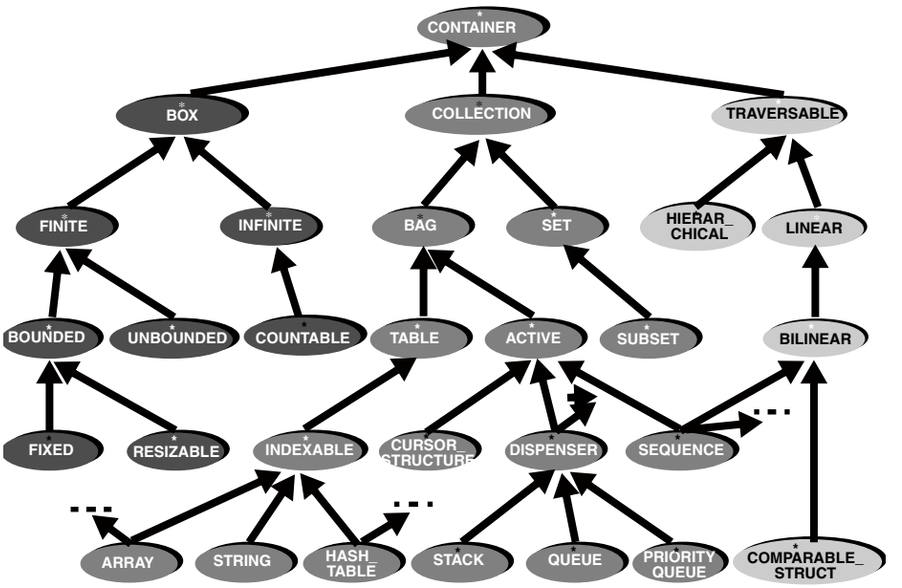


Figure 2: EiffelBase classes, topmost inheritance hierarchy

An obvious objection to the choice of EiffelBase is that industrial users may be more interested in coarser-grain components (EJB, CORBA, COM, .NET) covering — say — print drivers, web services or payroll records rather than stacks and linked lists. But several reasons actually make components such as the EiffelBase classes actually very useful for such a study:

- These components are used extensively in all applications. Bringing them to a high degree of quality is by itself an important achievement.
- They raise difficult problems, providing a microcosm of the more general issues of component-based development.
- The underlying theory has been explored in depth, and so is ready for integration in a proof framework.
- They are a required step: we can hardly hope to provide proofs of more complex, application-specific components until we know how to tackle these. If we can't prove the contracts of a list class, we won't be able to prove a Grid control.
- Finally, as noted, this is not the *whole* approach, simply one of its facets, complemented for more immediate industry needs by the "low road" techniques mentioned above.

So they appear a proper choice for this foundational work, a first step towards the development of general techniques for proving properties of different component kinds, fine-grained and coarse-grained.

5 A Reusable Effective Class: *STACK*

Let's take a look at a typical class and see what it would mean to "prove it". *FIXED_STACK* is one of the implementations of *STACK*, through arrays. Along with *ARRAYED_STACK*, *LINKED_STACK* and other implementations, it inherits from the deferred (abstract) class *STACK*:

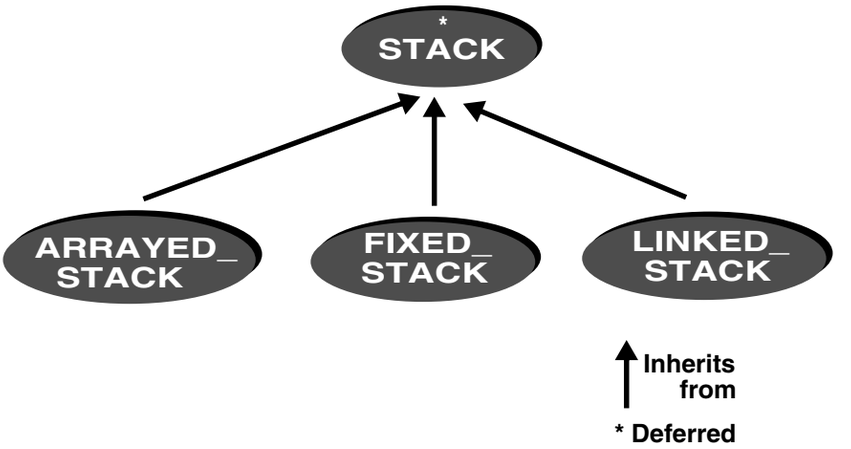


Figure 3: Multiple implementations through inheritance

In its general form, including of course the contracts, *FIXED_STACK* looks like this:

```
class FIXED_STACK [G] inherit

        STACK [G]
        ARRAY [G]
                rename item as array_item, put as array_put,
                        count as capacity
                end

feature -- Access
        count: INTEGER
                        -- Number of stack items

        item: G is
                        -- Top element
                require
                        not_empty: count > 0
                do
                        Result := array_item (count)

                end
```

```
feature -- Element change
    put (x: G) is
                -- Push x to top of stack.
        require
            not full: count < capacity
        do
            count := count + 1
            array_put (x, count)
        ensure
            pushed: item = x
            one_more: count = old count + 1
        end
    remove is
                -- Pop top element.
        require
            not_empty: count > 0
        do
            count := count – 1
        ensure
            removed: count = old count – 1
        end
invariant
    meaningful_count: count >= 0
    bounded_by_capacity: count <= capacity
end
```

A number of details have been omitted as compared to the actual class in the EiffelBase delivery:

- Assertion inheritance: most of the contracts are actually in the higher-level, deferred class *STACK*; class *FIXED_STACK* inherits them without having to reproduced them. What is shown here is close to its "flat form" rather than its actual text.
- Several features.
- Creation procedures.
- Some inheritance details.
- The Indexing clause.
- Array resizing (found in *ARRAYED_STACK*): in general, EiffelBase classes avoid putting strict limits on size, and instead silently resize the representation if the contents outgrow it. Here we have chosen the fixed-size variant since it leads to a simple and interesting contract.

6 Introducing a Model

The first question to ask is: What does it mean to "prove" *FIXED_STACK*?

The accurate phrasing is of course that we are interested in proving the *correctness* of the class with respect to particular *specification properties*.

> Like the rest of the literature this discussion casually talks about "proving a software element", but this is just an abbreviation for the notion of proving that the element possesses explicitly specified properties.

Thanks to the presence of contracts, these properties are already present in the text, in the form of preconditions (**require**), postconditions (**ensure**), class invariants (**invariant**). We have to prove that the implementation satisfies these contracts: any routine called with the precondition satisfied will terminate and ensure the precondition; in addition, if it is called from outside the class, it will preserve the invariants.

To talk about such correctness proofs will we use a conceptual *model* of the corresponding structure. We consider, for example, that a stack is conceptually representable as a sequence, where the push operation (*put*) happens at the end:\

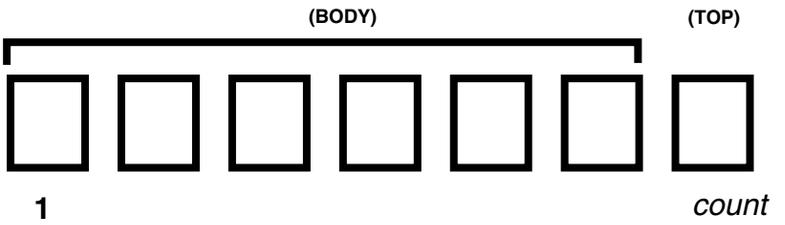


Figure 4: Modeling a stack as a sequence

Such a model plays no role in the class implementation; it is also of no significance to users of the class, who still view it in purely abstract terms, defined by official features (*put*, *remove*, *item* etc.) according to the principles of abstract data types. But it will help us for the proofs.

Mathematically, a sequence is a function from an integer interval:

$$SEQUENCE\,[G] \triangleq 1..count \rightarrow G$$

for some *count*. *s* (*i*) is the *i*-th item of *s*.

We may note here that the choice of a model is independent of the choice of implementation, such as *FIXED_STACK* rather than *ARRAYED_STACK* or *LINKED_STACK*. It can be done at the level of the deferred class *STACK*, their common ancestor. This opens up the possibility of doing part of the proof at the deferred level, an interesting prospect since it means that we can use inheritance for reuse and factorization in the *software proving* process like we do in the *software construction* process. Let us see how the deferred class — itself somewhat simplified too — looks:

```
deferred class
    STACK [G]

feature -- Access
    count: INTEGER
            -- Number of stack items

    item: G is
            -- Top element
        require
            not_empty: count > 0

        end

feature -- Element change

    put (x: G) is

            -- Push x to top of stack.

        require
            not_full: not full
        deferred

        ensure

            pushed: item = x
            one_more: count = old count + 1

        end

    remove is

            -- Pop top element.

        require
            not_empty: count > 0
        deferred
        ensure
            removed: count = old count – 1
        end

invariant

    meaningful_count: count >= 0

end
```

Abstract assertions

Now we can ask again the question "what does it mean to prove a class?" at the level of a deferred class such as *STACK*. The answer will be obtained by introducing the model (a sequence in our case) at that level.

7 Mathematical Classes

To proceed we need a better definition of basic model elements such as sequences. They will be described using simple set-theoretical concepts: Sets, Relations, Functions, Sequences (Z or B style).

For simplicity and consistency, it is useful to express such a notion in a form that looks like an Eiffel class:

```
deferred class
    SEQUENCE [G]

feature -- Access
    count: INTEGER
            -- Number of sequence items

    item (i: INTEGER): G is
            -- Element at index i
        require
            in_bounds: i >= 1 and i <= count
        deferred
        end

    last: G is
            -- Element of highest index
        require
            not_empty: count > 0
        deferred
        ensure
            definition: Result = item (count)
        end

    first: G is
            ... Similar ...

    subsequence (i, j: INTEGER): SEQUENCE [G] is
            -- Sequence made of elements from i to j, if any
        require
            first_in_bounds: i >= 1 and i <= count

            second_in_bounds: j >= 1 and j <= count
        deferred
        ensure
            Result.count = max (j – i + 1, 0)
        end
```

```
feature -- Element transformers
        extended (x: G): SEQUENCE [G] is
                -- Sequence extended with x at end.
            deferred
            ensure
                one_more: Result.count = count + 1
                added: Result.last = x
                rest_unchanged:
                  equal (Result.subsequence (1, count), Current),
            end
        head: SEQUENCE [G] is
                -- Sequence minus its first element
            require
                not_empty: count > 0
            deferred
            ensure
                removed: Result.count = count – 1
                rest_unchanged:
                    equal (Result, subsequence (1, count–1))
            end

... Similar declaration for tail ...
invariant
    meaningful_count: count >= 0
end
```

In spite of this programming-like appearance, however, such classes are purely mathematical objects. They correspond to the lowest level of the proof hierarchy suggested earlier:

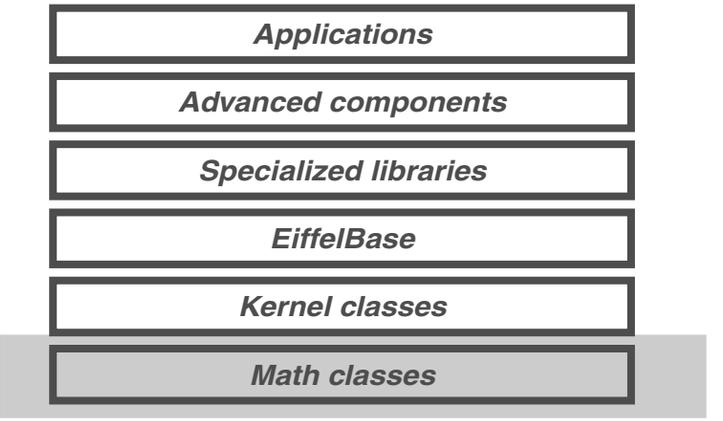


Figure 5: Place of mathematical classes

These classes are expressed in a subset of Eiffel called IFL (for Intermediate Functional Language), with strong restrictions: no procedures but only functions; no assignment to attributes; no modifications of any data structure. For example to represent sequence concatenation class *SEQUENCE* above uses a function *extended* that returns a new sequence, not a procedure *extend* that would modify the current sequence. In other words this is a purely applicative language, just a programming-like representation of pure mathematical concepts. We may view IFL as pure mathematics disguised in Eiffel syntax. Using a notation similar to that used for actual classes at higher levels of the hierarchy provides consistency and simplicity.

EiffelBase classes such as *STACK* and its variants rely on these IFL classes, through an intermediate level that we ignore for this discussion:

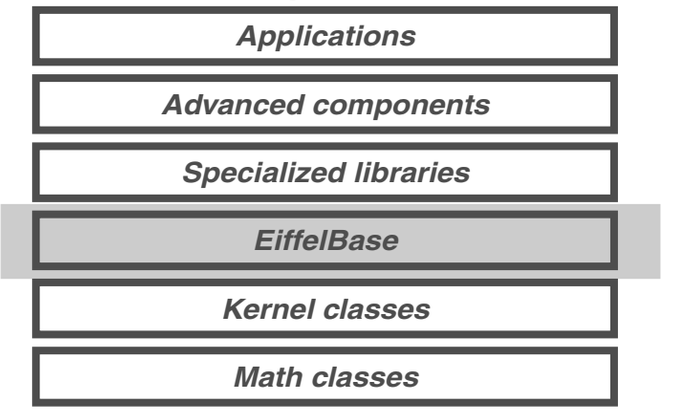


Figure 6: Place of data structure library

In the writing of class *STACK* we can now explicitly introduce the model. Using Eiffel techniques for selective export we export the corresponding features — and as a result the corresponding contracts — to a special class *SPECIFICATION* and to that class only. This means that they will not be visible in the official documentation ("contract form" or "short form") of the class.

```
deferred class
    STACK [G]

feature {SPECIFICATION} -- Specification
    model: SEQUENCE [G] is deferred end


feature -- Access
    count: INTEGER is
                -- Number of stack items
            deferred
            ensure
                same_count: Result = model.count
            end
```

```
    item: G is
            -- Top element
        require
            not_empty: count > 0
        deferred
        ensure
            is_last: Result = model.last

        end

feature -- Element change

    put (x: G) is

            -- Push x to top of stack.

        require
            not full: not full
        deferred
        ensure
            pushed: item = x
            one_more: count = old count + 1

            extended: model = old model.extended (x)

        end

remove is

            -- Pop top element.

        require
            not_empty: count > 0
        deferred
        ensure
            removed: count = old count – 1
            chopped_off: model = old model.head

        end

invariant

    meaningful_count: count >= 0
    model_exists: model /= Void

end
```

Abstract assertions

Model assertions

Our contracts now have two kinds of assertions, marked in the above extract: the original *abstract assertions*, part of the contracts for the class; and the new *model assertions*, introduced together with the model. We represent the model, in the class, by a new feature *model* (exported to *SPECIFICATION* only and hence not visible by normal clients); the model assertions are expressed in terms of *model*, referring here to features of the IFL class *SEQUENCE* as introduced earlier.

We have the first part of the answer to our general question of "what does it mean to prove a class?":

Proving a deferred class

To prove a deferred class is to prove that the model assertions imply the abstract assertions.

The following figure illustrates this process

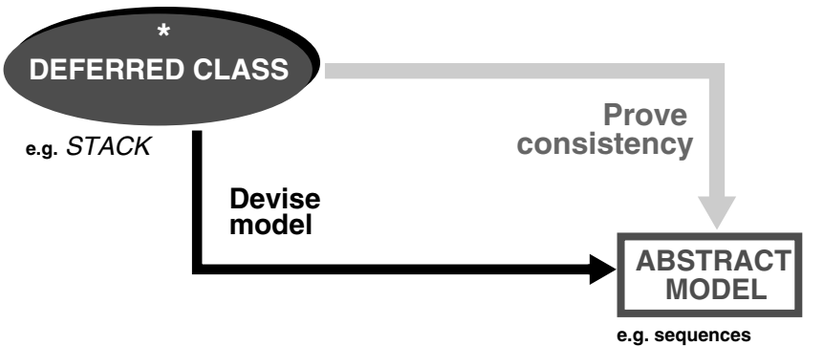


Figure 7: Proving a deferred class (partial view)

In the example of *STACK* and its *SEQUENCE* model, the proof is straightforward. For example, considering the postcondition of *put*, we set out to prove the first clause, *item* = *x*. The definition of *item* tells us that *item* is *model*.*last*. The model postcondition tells us that *model* = **old** *model*.*extended* (*x*), so *item* is **old** *model*.*extended* (*x*).*last*. But now the postcondition clause *definition* of *last* in class *SEQUENCE* tells us that this is indeed *x*.

The proof of the second postcondition clause, *count* = **old** *count* + *1*, is similar, using the postcondition of *extended* in *SEQUENCE*.

Note that these proofs of deferred classes really involve mathematical rather than programming properties, since there are no explicit assignment, feature call (the central O-O operation) or control structures.

In practice, it will be convenient to work not directly on the classes, but on mathematical versions that makes the manipulations easier, leading to the following variant of the above figure:

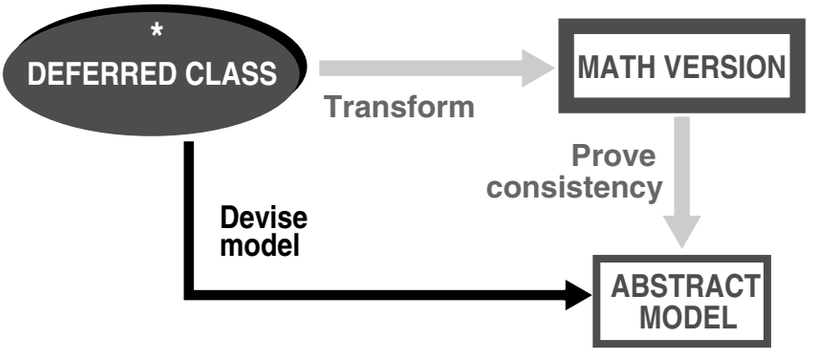


Figure 8: Proving a deferred class (full view)

8 Proving the Effective Class

We now come back to the effective class *FIXED_STACK*. It provides implementations of the features that were deferred in *STACK*, for example:

```
put (x: G) is
                -- Push x to top of stack.
        require
                not full: not full
        deferred
        ensure
                pushed: item = x
                one_more: count = old count + 1

                extended: model = old model.extended (x)

        end
```

The proof of the model assertions has been taken care of at the level of the deferred class. So what remains to be done is to check the consistency of the implementation vis-à-vis that first model:

Proving an effective class

To prove an effective class is to prove that the implementation satisfies the model assertions.

Of course the abstract assertions are those which matter for the clients. But by factoring out the choice of model at the level of the deferred class we have also factored out the proof of the abstract assertions; all that remains of interest at the level of individual effective classes is the model.

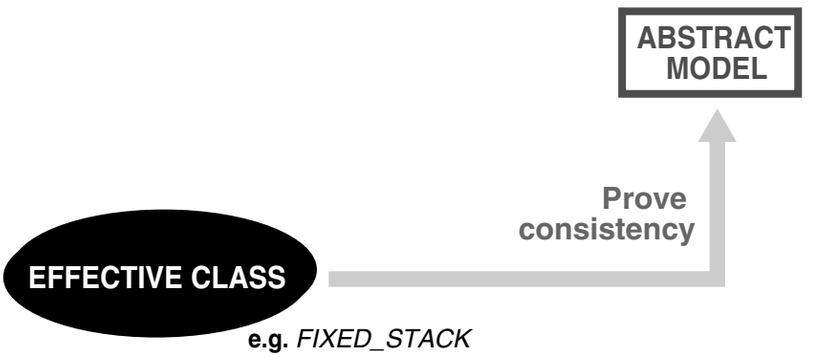


Figure 9: Proving an effective class (partial view)

In our example the proof is again straightforward, but requires an axiomatization of programming language constructs such as assignment, feature call and control structures, which the present discussion does not address.

Here too it's more convenient to work on a translation of the class into a mathematical form:

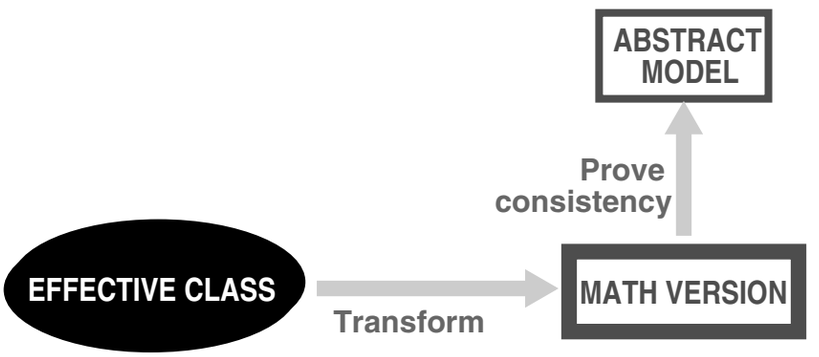


Figure 10: Proving an effective class (full view)

9 The Overall Scheme

The following figure combines the preceding ones to present the overall proof scheme:

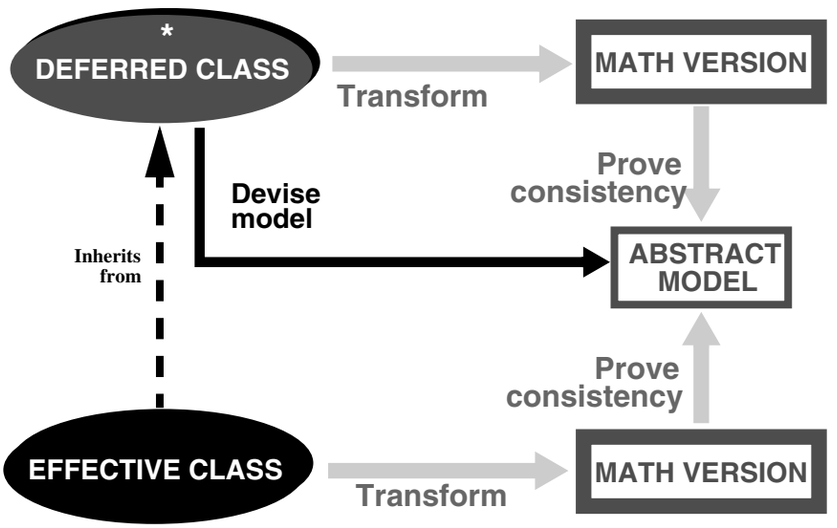


Figure 11: Scheme for proving classes

10 Issues to Be Addressed

The approach outlined here only describes a general scheme. A number of important details remain to be filled in:

- Choice of a mathematical model for proving class properties. Translating into mathematics avoids the symbol manipulation problems that often plague proof efforts.
- Finding the right mix between axiomatic and denotational approaches to modeling the software concepts involved.
- Handling the semantics, in particular (at the level of effective classes) for advanced language constructs.
- Adapting the theory to the form required by the chosen proof engine. Given the amount of proof work involved a fundamental requirement of the approach described here is mechanical support for proofs.
- Scaling up to bigger components.
- Exploring the possibility ot synthesizing the model automatically in some cases, instead of having to invent it for every class.

11 Strategy

Because of the number of concepts involved in a full-scale library such as EiffelBase (used daily by numerous applications, including large mission-critical systems) we cannot hope to tackle the full current library at once. A related issue is the number of programming language mechanisms that must be modeled.

To avoid an "all-or-nothing" approach, the strategy starts with elementary versions of both the language and the library, working its way up on both sides by introducing advanced language constructs and advanced library concepts one by one, and proving everything correct at each step.

We hope that this strategy will lead us to a fully proved and practically usable version of a contract-equipped library of fundamental computing structures.

References

[1] Jean-Raymond Abrial: *The B Book*, Cambridge University Press, 2002

[2] Felix Bachmann, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, Kurt Wallnau: *Technical Concepts of Component-Based Software Engineering*, Software Engineering Institute CMU/SEI-2000-TR-008, 2000, available at http://www.sei.cmu.edu/publications/documents/00.reports/00tr008.html.

[3] Bertrand Meyer, *Reusability*: *the Case for Object-Oriented Design*, in IEEE *Software*, vol. 4, no. 2, March 1987, pages 50-62. Also in: *Selected Reprints in Software*, ed. M. Zelkowitz, IEEE Press, 1987; *Software Reusability*, ed. T. Biggerstaff, Addison-Wesley, 1988; *Object-Oriented Computing*, IEEE Press, 1988.

[4] Bertrand Meyer: *Applying "Design by Contract"*, in IEEE *Computer*, 25, 10, October 1992, pages 40. Also in *Object-Oriented Systems and Applications*, ed. David Rine, IEEE Computer Press, 1994.

[5] Bertrand Meyer: *Eiffel*: *The Language*, Prentice Hall, second printing, 1991.

[6] Bertrand Meyer: *Reusable Software*: *The Base Object-Oriented Libraries*, Prentice Hall, 1994.

[7] Bertrand Meyer: *Object-Oriented Software Construction*, *second edition*, Prentice Hall, 1997.

[8] Bertrand Meyer, Christine Mingins and Heinz Schmidt: *Providing Trusted Components to the Industry*, in IEEE *Computer*, vol. 31, no. 5, May 1998, pages 104-105.

[9] Bertrand Meyer: *Contracts for Components*, in *Software Development*, vol. 8, no. 7, July 2000, pages 51-53.

[10] Bertrand Meyer: *Towards a Component Quality Model*, working paper at www.inf.ethz.ch/~meyer.

[11] Bertrand Meyer: *Proving Program Pointer Properties*, working paper at www.inf.ethz.ch/~meyer.

[12] Robert F. Stärk, Egon Borger and Joachim Schmid: *Java and the Java Virtual Machine*: *Definition*, *Verification*, *Validation*, Springer-Verlag, 2001.

[13] Clemens Szyperski: *Component Software*: *Beyond Object Oriented Programming*. Addison-Wesley, 1998.

Design by Contract is a trademark of Eiffel Software.

Mobile UNITY Schemas for Agent Coordination

Gruia-Catalin Roman and Jamie Payton

Department of Computer Science and Engineering,
Washington University in St. Louis
Campus Box 1045, One Brookings Drive, St. Louis, MO 63130-4899, USA
{roman,payton}@cse.wustl.edu

Abstract. Mobile UNITY refers to a notation system and proof logic initially designed to accommodate the special needs of the emerging field of mobile computing. The model allows one to define units of computation and mobility and the formal rules for coordination among them in a highly decoupled manner. In this paper, we reexamine the expressive power of the Mobile UNITY coordination constructs from a new perspective rooted in the notion that disciplined usage of a powerful formal model must rely on formally defined schemas. Several coordination schemas are introduced and formalized. They examine the relationship between Mobile UNITY and other computing models and illustrate the mechanics of employing Mobile UNITY as the basis for a formal semantic characterization of coordination models.

1 Introduction

Mobile UNITY [11, 19] is a formal model developed with mobility in mind and out of the desire to accommodate a new generation of systems that exhibit physical mobility of hosts and logical mobility of code (e.g., code on demand, remote evaluation, and mobile agent paradigms). Mobile UNITY inherits the simplicity of the original UNITY [4] and specializes its parent model in two important ways. First, it imposes a program structure that emphasizes decoupling and modularity. Systems (multiple cooperating programs in the parlance of UNITY) are structured in terms of program descriptions (types), program instances, and interactions. The last part defines the rules by which information is exchanged among programs which otherwise cannot communicate with each other—in a departure from UNITY, all variable names are local and only the interaction policy can refer to variables across program boundaries. Second, Mobile UNITY introduces several new constructs, including the reactive statement, which represents the boldest departure from its UNITY origins. While the term might be suggestive of event processing constructs in other models, the semantics of Mobile UNITY reactive statements are unique. The reactive statements are triggered not by events but by system state and have the power to alter the current state until a new configuration is reached in which no reactive statements are enabled.

The new model of concurrency inherits much of the notation and the proof logic directly from UNITY but makes the transition to mobility straightforward,

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 126–150, 2003.
© Springer-Verlag Berlin Heidelberg 2003

with no impact on program structure or proof logic. The typical application of Mobile UNITY to mobility is actually only a schema definition, i.e., a syntactic restriction on the general model. In this schema, each program is required to have a distinguished variable that denotes the current location of the program in some logical or physical space, and all interactions are conditional on the relative locations of the programs making up the system.

The Mobile UNITY approach to mobility is indeed different from that of Mobile Ambients [3], π-calculus [14], or mobile agent systems (e.g., LIME [16], MARS [2], D'Agents [7], etc.). Mobile Ambients structures space hierarchically and limits movement to conform to this hierarchical organization; the structure of the space is actually the structure of the system being described and can change with it. The π-calculus is a process algebra that allows for the creation of new unique channel names and for passing such names among processes for the purpose of establishing private communication channels. As a result, mobility is reduced to a restructuring of the communication structure, i.e., space and movement do not have a direct representation in the model. Mobile agent systems vary greatly but typically address the mobility of agents across connected hosts and the mechanics of coordination among them; the setting is generally that of logical mobility (LIME [16] is an exception in this respect, as it accommodates both logical and physical mobility), and space is assumed to be predefined. In Mobile UNITY space can be arbitrary, location is part of a program's state, and movement is reduced to value assignment to the location variable. Interactions among programs can be specified to meet the precise needs of each application domain.

The goal of this paper is to explore the richness of the model by examining a range of schemas that can adapt the basic Mobile UNITY model for a varied set of purposes. The expressive power of Mobile UNITY has been previously tested as part of a series of studies that demonstrated its ability to construct novel coordination constructs (e.g., transient and transitive variable sharing) [19, 11], to express clock-based synchronization [12], to specify and verify communication protocols (e.g., Mobile IP) [12], to specify and reason about code mobility paradigms [18], and to function as a semantic model for coordination constructs offered by new middleware for mobility. While building on earlier experience with Mobile UNITY, this paper seeks to distill the knowledge we gained and present its essence by reduction to a set of simple schemas that might assist in future efforts to apply Mobile UNITY to novel settings.

The Mobile UNITY model is presented in Section 2. The importance of schemas is highlighted in Section 3, where it is demonstrated that the treatment of mobility in Mobile UNITY can be defined as a schema. Sections 4 through 7 introduce new schemas for mobile computing and coordination. While the coordination mechanisms discussed in these sections relate directly to specific formal models or systems known in the literature on coordination or mobility, no effort is made to detail how a model or system might be simulated in Mobile UNITY, as our interest is in defining interesting coordination schemas rather than putting forth claims of universality. In each case, the origins of the schema

will be acknowledged, but its formalization will be abstract and minimalist. Finally, conclusions are presented in Section 8.

2 The Essence of Mobile UNITY

In this section, we give a brief introduction to the Mobile UNITY [11, 19] model, first describing the notation and associated proof logic, then applying the model to a simple example.

Notation. As in UNITY, the key elements of program specification are variables and assignments. Programs are simply sets of conditional assignment statements, separated by the symbol []. Each statement is executed atomically and is selected for execution in a weakly fair manner—in an infinite computation, each statement is scheduled for execution infinitely often. A program description begins by introducing the variables being used in the **declare** section. Abstract variable types such as sets and sequences can be used freely. The **initially** section defines the allowed initial conditions for the program. If a variable is not referenced in this section, its initial value is constrained only by its type. The heart of any Mobile UNITY program is the **assign** section consisting of a set of labeled conditional assignment statements of the form:

$$label :: var_1, var_2, \ldots, var_n := expr_1, expr_2, \ldots, expr_n \textbf{ if } cond$$

where the optional *label* associates a unique identifier with a statement. The guard *cond*, when false, reduces the statement execution to a skip.

As in UNITY, Mobile UNITY also provides quantified assignments, specified using a three-part notation:

$$label :: \langle \| \; vars : condition :: assignment \rangle$$

where *vars* is a list of variables, *condition* is a boolean expression that defines a range of values, and *assignment* is an assignment statement. For every instance of the variables in *vars* satisfying the *condition*, an *assignment* statement is generated. All generated assignments are performed in parallel. (This three part notation is used for other operations besides quantified assignment. For example, the $\|$ can be replaced with a '+', and all generated expressions are added together and a value is returned). Though not provided in the original model, the nondeterministic assignment [1] proved to be useful in many formalizations. A nondeterministic assignment statement such as $x := x'.Q$, assigns to x a value x' nondeterministically selected from among the set of values satisfying the predicate Q.

In addition to the types of assignment statements described above, Mobile UNITY also provides a *transaction* (not present in UNITY) for use in the **assign** section. Transactions capture a form of sequential execution whose net effect is a large-grained atomic state change (in the absence of reactive statements, which are explained later in this section). A transaction consists of a sequence

of assignment statements which must be scheduled in the specified order with no other statements interleaved in between. The notation for transactions is:

$$label :: \langle s_1; s_2; \ldots; s_n \rangle$$

The term *normal statement*, or simply *statement*, will be used to denote both transactions and the stand-alone assignments discussed earlier to contrast them with *reactive statements* introduced later in this section. As previously stated, normal statements are selected for execution in a weakly fair manner and executed one at a time. The guards of all normal statements can be strengthened without actually modifying the statement itself by employing *inhibit statements* of the form:

inhibit *label* **when** *cond*

where *label* refers to some statement in the program and *cond* is a predicate. The net effect is conjoining the guard of the named statement with the negation of *cond* at runtime, thus inhibiting execution of the statement when *cond* is true.

A construct unique to Mobile UNITY is the *reactive statement*:

s **reacts-to** *cond*

where s is an assignment statement (not a transaction) and *cond* is a predicate. The basic idea is that reactions are triggered by any assignment establishing the reactive condition *cond*. The semantics are a bit more complex since a program (or a *system*, which will be defined later) can contain many reactive statements. Operationally, one can think of each assignment (appearing alone or as part of a transaction) as being extended with the execution of all defined reactions up to such a point that no further state changes are possible by executing reactive statements alone. More formally, the set of all reactive statements forms a program $\Re$ that is executed to fixed point after each atomic state change by assignments appearing alone or within a transaction. Clearly, $\Re$ must be a terminating program. The result is a very powerful construct that can easily capture the effects of interrupts, dynamic system reconfigurations, etc.

Illustration. A sample Mobile UNITY program is shown in Figure 1. The program text specifies the actions of a baggage cart that moves along a track, loading at one end of the track and unloading at the other end. The program *Cart* defines variables y and λ of type **integer** in the **declare** section; y represents the size of the cart's current load, and λ can be thought of as the cart's location on the track. The **initially** section states that the cart is empty at the start of execution. Note that λ is not given a value in the initially section; λ can take on any value of type integer at the beginning of program execution. The **assign** section of *Cart* illustrates the use of several Mobile UNITY constructs.

Statement *load* is a simple conditional non-deterministic assignment statement that places a load in the cart (represented by the non-deterministic choice of a positive integer) if the cart is located at position 0 and is empty. The statements *go_right* and *go_left* are simple assignment statements that update the

```
program Cart
  declare
    y, λ : integer
  initially
    y = 0
  assign
    load        :: y := y'.(y' > 0) if λ = 0 ∧ y = 0
  [] go_right   :: λ := λ + 1
  [] go_left    :: λ := λ − 1
  [] inhibit go_right when y = 0
  [] inhibit go_left when y ≠ 0
  [] unload     :: y := 0 if λ = N ∧ y ≠ 0
  [] λ := 0 reacts-to λ < 0
  [] λ := N reacts-to λ > N
end
```

Fig. 1. An example Mobile UNITY program

cart's location on the track. The first inhibit statement prevents the execution of the *go_right* statement when the cart is empty. Similarly, the next inhibit statement prevents the cart from moving left when the cart is not empty. The next statement, *unload*, assigns to y a value of 0 if the cart is not empty and the cart is located at position N, effectively emptying the cart. The two statements following the *unload* statement are reactive statements. In the first, the reactive statement is enabled when the cart is at a position less than 0. If after the execution of a normal statement in the program, this statement becomes enabled, the cart's position is updated to a legal position (position 0) on the track. Similarly, the second reactive statement, when enabled, will force the cart in a legal position on the track, position N.

Proof Logic. Mobile UNITY has an associated proof logic by and large inherited directly from UNITY. Program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program text or from other properties through the application of inference rules.

Basic safety is expressed using the **unless** relation. For two state predicates p and q, the expression p **unless** q means that for any state satisfying p and not q, the next state in the execution sequence must satisfy either p or q. There is no requirement for the program to reach a state that satisfies q, i.e., p may hold forever. Progress is expressed using the **ensures** relation. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement that guarantees the establishment of q if executed in a state satisfying p and not q. Note that the **ensures** relation is not itself a pure liveness property, but is a conjunction of a safety and a liveness property. The safety part of the **ensures** relation can be expressed as an **unless** property, and the existence of an establishing statement

can be proven with standard techniques. In UNITY, the two predicate relations are defined by:

$$p \textbf{ unless } q \equiv \langle \forall s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

$$p \textbf{ ensures } q \equiv (p \textbf{ unless } q) \wedge \langle \exists s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{q\} \rangle$$

where s is a statement in the program P.

The distinction between UNITY and Mobile UNITY becomes apparent only when we consider the manner in which we prove Hoare triples, due to the introduction of transactions and reactive statements. For instance, in UNITY a property such as:

$$\{p\} s \{q\} \textbf{ where } s \textbf{ in } P$$

refers to a standard conditional multiple assignment statement s exactly as it appears in the text of the program P. By contrast, in a Mobile UNITY program we will need to use:

$$\{p\} s^* \{q\} \textbf{ where } s \in \aleph,$$

and $\aleph$ denotes the normal statements of P while s^* denotes a statement s modified to reflect the guard strengthening caused by inhibit statements and the extended behavior resulting from the execution of the reactive statements in the reactive program $\Re$ consisting of all reactive statements in P. The following inference rule captures the proof obligations associated with verifying a Hoare triple in Mobile UNITY under the assumption that s is not a transaction:

$$\frac{p \wedge \iota(s) \Rightarrow q, \{p \wedge \neg \iota(s)\} s \{H\}, H \mapsto (FP(\Re) \wedge q) \text{ in } \Re}{\{p\} s^* \{q\}}$$

For each non-reactive statement s, $\iota(s)$ is defined to be the disjunction of all **when** predicates of inhibit clauses that name statement s. Thus, the first part of the hypothesis states that if s is inhibited in a state satisfying p, then q must be true of that state also. $\{p \wedge \neg \iota(s)\} s \{H\}$ (from the hypothesis) is taken to be a standard Hoare triple for the non-augmented statement s. H is a predicate that holds after execution of s in a state where s is not inhibited. It is required that H leads to fixed-point and q in the reactive program $\Re$.

For transactions of the form $\langle s_1; s_2; \ldots s_n \rangle$ the following inference rule can be used before application of the one above:

$$\frac{\{a\} \langle s_1; s_2; \ldots s_{n-1} \rangle^* \{c\}, \{c\} s_n{}^* \{b\}}{\{a\} \langle s_1; s_2; \ldots s_n \rangle^* \{b\}}$$

where c may be guessed at or derived from b as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. This rule can be used recursively until we have reduced the transaction to a single sub-action. This rule may seem complicated, but it represents standard axiomatic reasoning for ordinary sequential programs, where each sub-statement is a predicate transformer that is functionally composed with others.

System Specification. So far, the notation and logic of Mobile UNITY have been discussed in terms of a single program. However, Mobile UNITY structures computations in systems consisting of multiple components and coordination rules that govern their interactions. Each component is a program with unique variable names. Programs are defined as instantiations of program types. Program type definitions are followed by a **Components** section that establishes the overall system configuration and some initialization parameters and by an **Interactions** section consisting of coordination constructs used to capture the nature of the data transfers among the decoupled component programs.

A **System** description begins by providing parameterized type definitions for the programs to be composed. A type definition of a program is simply program text that has a parameter used only to identify an instantiation of a program. Type definitions are similar to macros in that the textual type definition of a program can be substituted for a program instantiation anywhere within the **System**.

In the **Components** section of a system, component programs are instantiated using the name of a type definition and a parameter value to identify the instantiated program. The **Components** section assumes a form such as:

$$programA(1) [] programA(2) [] programB(1)$$

where *programA(i)* and *programB(j)* are type definitions in the system, and *programA(1)*, *programA(2)*, and *programB(1)* are the desired program instantiations.

Instantiated programs making up a **System** in Mobile UNITY have disjoint namespaces. The separate namespaces for programs hide variables and treat them as internal by default, instead of being universally visible to all other components as is the case in UNITY program composition. Formally, uniqueness of variable names in Mobile UNITY systems is achieved by implicitly prepending the name of the component to that of each variable, e.g., $programA(1).x$, $programB(1).x$. This facilitates modular system specification and impacts the way program interactions are specified for those situations where programs must communicate. Coordination among programs in Mobile UNITY is facilitated by defining rules for program interaction in the **Interactions** section of a system.

The **Interactions** section of the system specification defines inter-process communication. As mentioned previously, programs in Mobile UNITY cannot interact with each other in the same style as in UNITY (by sharing identically named variables) because they have distinct namespaces. Instead, special constructs must be provided to facilitate program interaction. These rules must be explicitly defined in the **Interactions** section of a system, using fully-qualified variable names. Since in mobile computing systems, interaction between components is transient and location-dependent, the **Interactions** section often restricts communication based on variables representing location information. Reactive statements, inhibit statements, and assignment statements can appear in the **Interactions** section. Here, however, references to variables that cross program boundaries are permitted.

Illustration. Figure 2 shows a system called *BaggageTransfer*. It is based upon a restructuring of the earlier *Cart* program designed to separate the cart, loading, and unloading actions. Three types of components are used: *Cart(k)*, *Loader(i)*, and *Unloader(j)*. Each program type is parameterized so as to allow for the creation of multiple instances of the same type.

System *BaggageTransfer*
program *Cart(k)*
declare
y, λ : integer
initially
$y = 0$
assign
go_right $:: \lambda := \lambda + 1$
[] *go_left* $:: \lambda := \lambda - 1$
[] **inhibit** *go_right* **when** $y = 0$
[] **inhibit** *go_left* **when** $y \neq 0$
[] $\lambda := 0$ **reacts-to** $\lambda < 0$
[] $\lambda := N$ **reacts-to** $\lambda > N$
end

program *Loader(i)*
declare
x : integer
initially
$x = 0$
assign
load $:: x := x'.(x' > 0)$
end

program *Unloader(j)*
declare
z : integer
initially
$z = 0$
assign
unload $:: z := 0$
end

Components
$Cart(1)$ [] $Cart(2)$
[] $Loader(1)$ [] $Unloader(1)$

Interactions [1]
Cart(k).y, *Loader(i).x* := *Loader(i).x*, 0
when *Cart(k).y* = 0
$\wedge$ *Loader(i).x* $\neq 0$
$\wedge$ *Cart(k).*$\lambda = 0$
[] *Cart(k).y*, *Unloader(j).z* := 0, *Cart(k).y*
when *Cart(k).y* $\neq 0$
$\wedge$ *Unloader(j).z* $= 0$
$\wedge$ *Cart(k).*$\lambda = N$

end *BaggageTransfer*

Fig. 2. An example Mobile UNITY system

Cart(k) defines a program in which a baggage cart is moved along a track. As before, the movement of the cart depends on the value of the program variable y, which represents the weight of the current baggage in the cart. Notice that the program type definition contains no statement in which y is explicitly assigned. *Loader(i)* defines a program in which a variable x is non-deterministically assigned a value, presumably defining a baggage weight to be loaded. *Unloader(j)* defines a program in which a variable is assigned a value of 0.

The **Components** section instantiates the component programs in the *BaggageTransfer* System. Two carts (*Cart(1)* and *Cart(2)*) are created along with

[1] Though its semantics are identical to those of the **if** keyword, the **when** keyword is used for emphasis in the **Interactions** section of Mobile UNITY systems.

a single loader and unloader. The two carts are distinguished by the distinct values given to parameter k.

The **Interactions** section allows the carts, loader, and unloader program instantiations to work together to transport baggage. The first statement is an asynchronous value transfer conditional on the location of the cart and the status of the loader. Since all free variables are assumed to be universally quantified by convention, the statement describes the relationship between a typical loader and a typical cart, and so it applies to both carts. The load stored in *Loader(i).x* is transferred to the cart and stored in *Cart(k).y*. This will enable the cart to start its movement towards the unloader. In a similar fashion, the arrival of a cart at the right hand side of the track makes it possible for the load to be transferred from *Cart(k).y* to *Unloader(j).z*, later to be discarded as apparent in the code of the unloader.

As shown elsewhere [11], many different coordination constructs can be built out of the basic constructs presented so far. Among them, one of particular interest is transient and transitive variable sharing, denoted by $\approx$. For instance, the code below describes an interaction between a cart and an inspector where the cart and the inspector share variables y and w as if they denoted the same variable, when co-located. At the point when the cart and inspector become co-located, the shared variable is given the value of the cart's y variable as specified by the **engage** clause. When the cart and inspector are no longer co-located, the cart's y variable retains the value of the shared variable and the inspector's w variable is set to 0, as stated in the **disengage** clause.

$$\begin{array}{ll} \textit{Cart(k).y} \approx \textit{Inspector(q).w} & \textbf{when } \textit{Cart(k).}\lambda = \textit{Inspector(q).}\lambda \\ & \textbf{engage } Cart(k).y \\ & \textbf{disengage } Cart(k).y,\ 0 \end{array}$$

Proof Logic Revisited. The entire system can be reasoned about using the logic previously presented because it can easily be re-written as an unstructured program with the name of each variable and statement expanded according to the program in which it appears, and with all statements merged into a single **assign** section. In other words, the system structuring represents solely a notational convenience with no deep semantic implications.

3 Mobile UNITY as a General Schema for Mobility

The features that differentiate Mobile UNITY [11] from the original UNITY [4] model have been the result of a careful analysis of what is necessary to model mobile systems. Yet, the presentation so far deliberately avoided showing these constructs in the context of mobility. This is because the new features, while needed to capture mobility, can be used in a wide range of settings and for a variety of purposes. In this section, we specialize the general notation to one form of mobility and coordination representative of much of the published work on

Mobile UNITY. We accomplish this by defining a schema for mobility. Other schemas specialized for other situations and related to existing models of coordination will be described in following sections.

A schema is typically defined as a syntactic (sometimes semantic) pattern which restricts the use of language constructs to specific forms that have desirable properties. Efficient implementation or accurate modeling of key application concerns are properties one may seek in defining a particular schema. In this section and in the remainder of this paper, we focus on the latter. More precisely, in this section we desire to restrict the model in a manner that allows one to directly capture systems whose components can move freely through a space and interact with each other in a location-dependent manner. Programs are defined to be the basic unit of mobility, modularity, and execution. This is a natural choice in Mobile UNITY and, fortunately, places no undue burden on the modeling process because programs can be of arbitrary complexity. Both fine-grained mobility (e.g., single statements) and coarse-grained mobility (e.g., whole components) may be expressed simply by varying the size of the programs being used. The use of program types facilitates compact representation of systems consisting of large numbers of mobile components. For now, we impose no restrictions on the size of the program code, the functions it performs, or the number of components that are being instantiated. In a given application setting, however, such restrictions may prove to be highly profitable, e.g., when one considers the case of very small devices such as sensors dedicated to evaluating one single local environmental condition, such as temperature.

Because programs are expected to be mobile, a mechanism must be introduced to capture the notion that a given component is present at a specific location and that it can move from one location to another. We choose to model location as a distinguished variable which is required to appear in all programs. Conventionally, this variable is named λ. Figure 3 depicts a new version of the *BaggageTransfer* system presented earlier, slightly modified to conform with the mobility schema presented in this section. The first thing to observe is the slight change in notation. The distinguished variable λ is pulled into the program type declaration. The resulting notation is merely a mechanical transformation designed to enforce the distinguished nature of the variable λ. Furthermore, the initialization of λ, if necessary, is relegated to the **Components** section.

By having an explicit representation of the program location as part of its state, mobility is reduced to changes in the value of λ. The type of λ is determined by the specific way in which space is modeled. In the example shown in Figure 3, we assume a discrete linear space over the range 0 to N. Other notions of space can be used with equal ease. When modeling physical movement, a latitude and longitude pair may be appropriate in defining a point in space. Logical mobility may entail the use of host identifiers. Spaces may be uniform and bounded, may be undefined in certain regions, or may extend to infinity. The operations permitted for use in changing λ are specified implicitly in the definition of the space. As before, in Figure 3 the location of the cart is incremented or decremented one unit at a time, thus faithfully representing the nature of

System *BaggageTransfer*
  program *Cart(k)* **at** λ
    ...
  end

  program *Loader(i)* **at** λ
    ...
  end

  program *Unloader(j)* **at** λ
    ...
  end

Components
  Cart(1) [] *Cart(2)* [] *Loader(1)* at 0 [] *Loader(2)* at $N/2$
[] *Unloader(1)* at N [] *Unloader(2)* at $3N/4$

Interactions
  Cart(k).y, *Loader(i).x* := *Loader(i).x*, 0
    when $Cart(k).y = 0 \wedge Loader(i).x \neq 0 \wedge Cart(k).\lambda = Loader(i)\lambda$
[] *Cart(k).y*, *Unloader(j).z* := 0, *Cart(k).y*
    reacts-to $Cart(k).y \neq 0 \wedge Unloader(j).z = 0 \wedge Cart(k).\lambda = Unloader(j).\lambda$

end *BaggageTransfer*

Fig. 3. Sample application of the general mobility schema

discrete but contiguous linear movement. When the space being modeled has a specific structure, mobility requires appropriate constraints. For instance, if the space is defined as a graph, it is reasonable to expect that movement takes place along edges in the graph. In other cases, we may prefer to allow a program to change location by simply moving to any reachable node in the graph if the passage through intermediary nodes results in no local interactions. In the *BaggageTransfer* example presented in Figure 3, changes to the cart's λ variable are restricted so that the cart can only move along positions on the track. It is important to note that, by reducing movement to value assignment, the proof logic naturally covers mobility without necessity for extensions.

The manner in which programs use the location variable, in turn, may induce several subschemas. A program may be location-oblivious in the sense that it never refers to λ anywhere in its code (except for its compulsory declaration). In such cases, mobility is external to the program, however, its behavior may be affected by mobility indirectly, i.e., by the coordination rules defined in the **Interactions** section. Location-aware programs refer to the value of λ and alter their behavior based on their current location. Finally, location-controlling programs actually modify their location explicitly in their code. In Figure 3, the loader and unloader programs are location-oblivious, and the cart programs are location-controlling.

Finally, we turn our attention to the **Interactions** section. It is intended to serve as a repository for the inter-program coordination rules. In general, statements present in the **Interactions** section can be of an arbitrary nature and are permitted to refer to variables owned by the individual programs contained within the system.

In the case of a mobile system, it is reasonable to assume that interactions are local. When mobile code is involved, interactions among programs take place whenever the components are co-located. In the presence of physical mobility, connectivity relies on wireless communication that is limited in range. Given such considerations, a reasonable restriction might be to require all statements appearing in the **Interactions** section to limit their effects only to pairs of variables in co-located programs. The co-location requirement can be a reasonable abstraction for situations in which all components within some limited region of the space can communicate with each other. Pairwise interactions may be imposed due to the nature of the underlying communication protocols (e.g., message passing). In Figure 3, both interactions are guarded by the requirement that the affected components are at the same location. In one case, whenever either cart is co-located with a loader, loading is possible but not required. In the other case, by using a reactive statement, a cart co-located with an empty unloader is guaranteed to unload.

At this point, all the essential features of this mobility schema are represented in Figure 3, but a slight generalization could result in a richer model. We achieve this by permitting the equality relation used among locations to be replaced by any arbitrary binary relation among points in space. For instance,

$$\textit{within_range}(A.\lambda,\ B.\lambda) \text{ or } \textit{reachable}(A.\lambda,\ B.\lambda)$$

would allow one to accommodate the notion of limited wireless transmission range or connectivity in wired networks, respectively.

In this paper, we are concerned with a particular class of mobile systems, one that emphasizes spatial and temporal decoupling among components. In coordination models, such decoupling facilitates component interactions (exhibiting various degrees of decoupling) that are more abstract than those provided by the typical communication layers. In the remainder of this paper, we will explore styles of coordination and Mobile UNITY's ability to capture their essential features in schemas.

4 Agent Mobility in Wired Networks

Agent systems represent a popular new mode of computing specifically designed to take advantage of the global space offered by the Internet. An agent is a code fragment that can move from site to site in purposeful pursuit of some task defined at its point of origin. The underlying space is a graph whose vertices denote servers willing and able to accept agents. Since Internet connectivity may be perceived to be reliable and universal, the edges in the graph represent accessibility to other sites. Each agent carries with it not only the code to be

executed, but also data and control state that affect its actions at each site. The movement from one site to the next may be autonomous (subjective) or initiated by the server (objective). Agents belonging to the same community of applications may interact with each other. In D'Agents [7], for instance, message passing, streams, and events are used to enable agents to communicate among themselves. Agent systems that stress coordination rather than communication tend to rely on tuple spaces, in the spirit of the original coordination modality proposed in Linda [6]. TuCSoN [17], MARS [2], and Limbo [5] are just a small sample of agent systems that employ tuple based coordination. They provide the basis for the schema proposed below.

In examining such systems, the following features capture their essence:

- agent mobility among servers
- admission control
- coordination by means of tuple spaces located on the server
- traditional tuple space operations, e.g., **out**(*tuple*), **in**(*pattern*), **rd**(*pattern*)
- augmentation of tuple space operations with reactions that extend the effects of the basic operations to include arbitrary atomic state transitions.

A coordination schema that enforces this design style will need to distinguish between agents and servers. Syntactically this can be done by substituting **Server** or **Agent** for the keyword **Program**, as needed. For instance, one can do this by means of a macro definition of the form:

Agent X ≡
    Program Agent_X

With this distinction, we can examine the different requirements for agent and server programs. The agent location is the location of one of the servers and the change in location can be accomplished by modifying λ to hold the value of some other server location, including the agent's home location. For reasons having to do with admission control, it is best to think of λ as holding a pair of values:

$\lambda \equiv$
    (current_location, desired_server_relocation)

and provide the agent with a single move operation:

$goto(\mathcal{S}) \equiv$
    $\lambda := (\lambda \uparrow 1, S)$

We use $var \uparrow n$ to denote the n^{th} element held by the record-like variable var. Since checking that the new location is valid requires comparing the agent's location with the location of a server, the actual move (i.e., the assignment to λ) is relegated to the **Interactions** section.

While the agent is present at a particular server, all interactions with the server and other agents take place by sharing a single tuple space owned by the server. The variable T could be used to represent such a tuple space (assumed

here to be a set of tuples) inside the agent with the **Interactions** section establishing the tuple sharing rules. Access to T is restricted to tuple space operations. The **out** operation simply adds a tuple to the set if the guard g is true:

$$\textbf{out}(t) \textbf{ if } g \equiv$$
$$\quad T := T \cup \{t\} \textbf{ if } g.$$

The **in** operation is blocking and removes a tuple matching some pattern p:

$$z = \textbf{in}(p) \textbf{ if } g \equiv$$
$$\quad \langle\, \theta : \theta = \theta'.(\theta' \in T \wedge match(\theta', p)) \wedge g :: z := \theta \| T := T - \{\theta\} \rangle$$

where we use the nondeterministic value selection expression $x'.Q$ to identify one suitable tuple. If none exists, the operation is reduced to a skip. Busy waiting is the proper modeling solution for blocking operations in the Mobile UNITY context. The **rd** operation is similar to an **in**, the only difference being that the returned tuple is not removed from the tuple space.

The server also has a location λ and a variable T, but its location cannot change. For the sake of uniformity, the server's location variable must hold a pair like the agent's location variable λ, but the two fields hold identical values. Since the server is stationary, it cannot change its λ variable, and the *goto* operation is not available. However, the server needs to be aware of the presence of agents at its location, either in order to refuse admission by sending an agent back before it can have any local effects or by forcing an agent to move elsewhere when conditions demand it. The presence of an agent could be made known to the server by introducing a new variable Q in both agents and servers. On the agent, the variable contains a tuple i that identifies that agent but no operations are available to access it. The **Interactions** section will ensure that all the variables Q of all the agents are shared with the server forming a set of all agents present at that location. (The server need not store its own identity in Q.) The server can discover the presence of agents by reading the contents of Q without being able to modify it. This can be accomplished by hiding Q inside an operation such as:

$$AG{:=}LocalAgents() \equiv$$
$$\quad AG := Q$$

Finally, the server may request an agent to move to some other location by employing an operation such as:

$$Move\ A \text{ to } S \equiv$$
$$\quad M := (A, S)$$

which places in the hidden variable M a request to move agent A to server S. The actual move is encoded in the **Interactions** section, which will determine the location of S, will use it to change the location of A, and clear the request from M.

The syntactic restrictions on agent and server code are complemented by coordination patterns built into the **Interactions** section. First, we must specify the sharing rules governing the variables T and Q. Using the transient and transitive variable sharing of Mobile UNITY, the sharing rules become:

$$S.T \approx A.T \textbf{ when } S.\lambda \uparrow 1 = A.\lambda \uparrow 1$$
$$\textbf{engage } S.T$$
$$\textbf{disengage } S.T., \emptyset$$

$$S.Q \approx A.Q \textbf{ when } S.\lambda \uparrow 1 = A.\lambda \uparrow 1$$
$$\textbf{engage } S.Q \cup A.Q$$
$$\textbf{disengage } S.Q - \{A.\iota\}, \{A.\iota\}$$

where we assume that the initial value of $A.Q$ is permanently saved in $A.\iota$, another hidden variable.

Mobility requests are handled by introducing reactive statements designed to extend the request (a local operation) with its actual processing (a global coordination action). For instance, the objective move operation requested by the server is transformed into an equivalent hidden subjective request:

$$A.\lambda := (A.\lambda \uparrow 1, S.\lambda \uparrow 1) \| S'.M := nil$$
$$\textbf{reacts-to } A.\lambda \uparrow 1 = S'.\lambda \uparrow 1 \wedge S'.M = (A, S).$$

This, in turn, results into two cases to consider: when A is accepted by the destination $\mathcal{S}$ and the move is carried out:

$$A.\lambda := (S.\lambda \uparrow 1, S.\lambda \uparrow 1) \textbf{ reacts-to } A.\lambda \uparrow 2 = S.\lambda \uparrow 1 \wedge admitted(A.Q, S)$$

and when the move is rejected and the agent is reactivated at its current location:

$$A.\lambda := (A.\lambda \uparrow 1, A.\lambda \uparrow 1) \textbf{ reacts-to } A.\lambda \uparrow 2 = S.\lambda \uparrow 1 \wedge \neg admitted(A.Q, S).$$

As an example, consider an inspector agent that moves among unloader service sites and computes the total number of packages that pass through the system. Each unloader is assumed to hold a local counter of packages. The inspector adds the local counter to its own and resets the local one. Once all sites are visited, the inspector agent returns home. Each site will reject any inspector agent that is not authorized to collect the data. By employing the schema presented in this section, the agent code for this example becomes:

program *Inspector(k)* **at** λ

always
$home_again = (\lambda = (home(k), home(k)))$

declare
...

initially
$\iota = (inspector.k, password(k))$
$[\!] \; Q = \{\iota\} \; [\!] \; T = \{\} \; [\!] \; N = 0 \; [\!] \; \lambda = (home(k), home(k))$

assign
$\langle goto(next_server(\lambda)); t := in(\langle counter, int : m\rangle);$
$N := N + t \uparrow 2;\ out(\langle counter, 0\rangle)\rangle$
$[\!] \; N := 0$ **reacts-to** $home_again$

end

One element still missing from the schema definition is the augmentation of tuple space operations with arbitrary extra behaviors. This can be accomplished by separating the initiation of an operation from its execution. An **in** operation, for instance, can be redefined as a request RQ which, in turn, can enable a programmer specified reaction on the server:

t:=**in**(p) **if** $g \equiv$
$\langle RQ := (id, in, p)$ **if** $g; t, T, tt := tt, T - \{tt\}, nil$ **if** $tt \neq nil\rangle$

$\langle \|\theta : \theta = \theta'.(\theta' \in T \wedge \ match(\theta', p)) :: tt := \theta\rangle$ **reacts-to** $RQ \uparrow 3 = p$

action **extends**$(\rho, \omega, \pi) \equiv$
action **reacts-to** $RQ \neq nil \wedge tt \neq nil \wedge \rho(RQ \uparrow 1) \wedge \omega(RQ \uparrow 2) \wedge \pi(RQ \uparrow 3)$
$\|\ RQ := nil$ **reacts-to** $RQ \neq nil \wedge tt \neq nil$

where (ρ, ω, π) specifies the criteria under which the **in** operation is extended. This illustration assumes one extension only, but it could be rewritten to accommodate multiple extensions to be applied in a nondeterministic order.

Since systems consist of components controlling local actions and interactions that extend their effects to other components, it is not suprising that the schema definition also seems to be structured along these lines: mostly syntactic restrictions of the component code (further refined by component type) and coordination patterns of a behavioral nature, restricted in scope solely to variables involved in the process of information sharing. It is this structuring of the schema definition that qualifies it as a *coordination schema*.

5 Agent Mobility in Ad Hoc Networks

In this section we explore the implication of extending the mobile agent paradigm to ad hoc networks. Ad hoc networks are formed when hosts come into wireless

contact with each other and communicate as peers in the absence of base stations and any wired infrastructure. In such settings, one can envision systems consisting of hosts that move through physical space and agents that reside on such hosts. Agents can coordinate application activities with other agents within reach, and also have the ability to move from one host to another when connectivity is available. One of the very few systems to offer these capabilities is LIME [16], which will be used as a model for the schema we explore in this section. The key features of LIME are as follows:

- each agent may create an arbitrary number of local tuple spaces, each bearing a locally distinct name;
- agents coordinate by sharing identically-named tuple spaces belonging to agents on connected hosts, i.e., each agent has access to all the tuples in such combined tuple spaces (called federated tuple spaces).

In Mobile UNITY, it is convenient to represent each tuple space as a pair of variables, one holding a name and the other storing the set of locally-owned tuples that are part of that tuple space, tuples the agent is willing to share with other agents. Consequently, the tuplespace sharing rule can be easily expressed as follows:

$$
\begin{array}{l}
A.X_T \approx B.Y_T \ \textbf{when}\ \mathit{connected}(A,B) \wedge A.X_N = B.Y_N \\
\qquad\qquad\qquad\qquad \textbf{engage}\ A.X_T \cup B.Y_T \\
\qquad\qquad\qquad\qquad \textbf{disengage} \\
\qquad\qquad\qquad\qquad \langle \text{set}\ t, C, Z : \mathit{connected}(A,C) \wedge A.X_N = C.Z_N \\
\qquad\qquad\qquad\qquad\quad \wedge\ t \in A.X_T \wedge\ t\ \mathit{owned\ by}\ C :: t \rangle, \\
\qquad\qquad\qquad\qquad \langle \text{set}\ t, C, Z : \mathit{connected}(B,C) \wedge B.Y_N = C.Z_N \\
\qquad\qquad\qquad\qquad\quad \wedge\ t \in B.Y_T \wedge\ t\ \mathit{owned\ by}\ C :: t \rangle
\end{array}
$$

where *connected* is defined in terms of reachability in the ad hoc network and the extensions $_N$ and $_T$ refer to names and sets of tuples, respectively. Upon connection, the engagement value is the union of all the connected identically-named tuple spaces, and, upon disconnection, the set of tuples is repartitioned according to the new connectivity pattern. However, in order to accomplish this, the concept of tuple ownership needs a representation; we assume that each tuple includes a current location field (an agent id, ι) which allows us to define:

$$
\begin{array}{l}
t\ \mathit{owned\ by}\ C \equiv \\
\quad t.loc = C.\iota
\end{array}
$$

In the above, we take the liberty to assume that fields in a tuple could be referenced by name. It is interesting to note the kind of hierarchical spatial organization emerging from this schema: hosts have locations in the physical space and their wireless communication capabilities can be abstracted by a reachability predicate, not shown but implied in the definition of *connected*; agents reside on hosts or servers in a manner similar to that shown in the previous section (for this reason, we do not repeat the details of agent movement even though now it

is conditional on the availability of connectivity); tuples reside on agents, a new logical space defined by the name of the tuple space combined with that of the agent.

Since tuples have a logical location, it becomes reasonable to consider the possibility of restricting operations on tuples to particular subspaces, and to entertain the notion of tuple movement. Actually, LIME offers both capabilities. For instance, **in** and **out** operations can be restricted to a specific agent location. More interestingly, **out** operations can be targeted to a particular location, i.e., the named tuple space of a particular agent. Since the agent may not be connected at the time, the tuple is augmented with a second location field that stores the desired destination. This is reminiscent of the agent mobility treatment from the previous section but with one important difference—the tuple will continue to reside locally until such time that migration becomes possible. Migration, immediate or upon the establishment of a new connection, is captured by an interaction of the form:

$$
\begin{array}{l}
A.X_T := \\
\quad \langle \text{set } t, B, Y : t \in A.X_T \wedge \ t.dest = B \wedge \ connected(A, B) \\
\qquad \wedge \ A.X_N = B.Y_N :: t[loc : B; dest : B] \rangle \\
\quad \cup \\
\quad \langle \text{set } t, B, Y : t \in A.X_T \wedge \ t.dest = B \wedge \ \neg(connected(A, B) \\
\qquad \wedge \ A.X_N = B.Y_N) :: t \rangle \\
\qquad \textbf{reacts-to } true
\end{array}
$$

where we use the notation $t[field_name : newvalue]$ to denote a modification of a particularly named field in tuple t.

Since the purpose of this paper is to explore coordination schemas, we refrain from including here all features of LIME and limit ourselves to noting that a complete formalization of LIME in terms of Mobile UNITY has been performed [15]. The features that were discussed in this section demonstrate the applicability of the model to an area of computing of growing importance, one that presents new challenges to the software engineering community.

6 Mobility in Malleable Program Structures

In some systems, the definition of space is the program itself. In Mobile Ambients [3], for instance, the program consists of environments called ambients that can be embedded within each other. Mobility takes place by altering the relation among ambients, which, for mobility purposes, are treated as single units. An ambient can exit its parent and become a peer with the parent; an ambient can enter a peer ambient; and an ambient can dissolve the domain boundary of a peer ambient. All these can be done only if the name of the relevant ambient is known. This is a way to model security capabilities. Other systems, such as MobiS [9], are more restrictive in terms of the range of operations provided for mobility while others, such as CodeWeave [10], may approach mobility at a much

finer level of granularity—in CodeWeave, single statements and single variables can be moved anywhere in the program structure where the latter is distributed across hosts and is hierarchical along the lines of block-structured programming languages.

The schema we describe in this section is directly inspired by Mobile Ambients. Key points of distinction will be related to fundamental differences between a process algebra and a programming notation that does not support dynamic process creation or scope restriction. To avoid possible confusion, we will use the term spatial domain, or simply domain, to refer to the analog of an ambient. The defining features of the resulting schema are:

- hierarchical structuring of the space in terms of embedded domains that reflect directly the overall structure of the system;
- protection enforcement via capabilities that rely on secret unique names;
- mobility in the form of localized restructuring of the system structure.

In Mobile UNITY, a system is simply a collection of programs. One way to organize it hierarchically and still allow for dynamic reconfiguration is to impose a partial order over the set of programs in a manner that corresponds to a tree having an imaginary root. A domain is defined in terms of all the programs that share a common parent, and the name of the parent can be used to uniquely designate that domain. This can be encoded by simply setting λ to refer to the $(program, parent)$ pair of names. An assignment of location values in the **Components** section defines the initial program structure. At the start, each program is given a unique name which, as explained later, can change over time. The program instance parameter can be used for this purpose. Below is an example of a well-formed **Components** section:

$A(1)$ **at** (1, 0)
$B(1)$ **at** (1.1, 1)
$C(1)$ **at** (1.2, 1)

where A, B, and C receive hidden distinct names 1, 1.1, and 1.2, respectively. The above establishes four domains: domain 0, which contains $A(1)$; domain 1, which contains the peer components $B(1)$ and $C(1)$; domain 1.1, which is empty; and domain 1.2, which is also empty. References to domain names will be needed in the programs. For this reason we assume that a distinguished variable ι provides each program with its own name, assumed to be unique. We assume, however, that λ (the pair consisting of ι and its parent, i.e., the domain name) is not directly accessible to the individual programs, i.e., the schema rules out statements that refer to λ in any way.

To enforce some sort of scoping constraints, we simply require that program to program communication be restricted only among peers. The type of communication is not important for the remainder of this presentation, but the reader should assume that it is available in the form of tuple space coordination or synchronous message exchange. One thing that is important is the fact that program/domain names can be passed among programs.

In the spirit of Mobile Ambients, we treat naming as the critical element of any security enforcement policy. Without exception, all operations entailing mobility involve a domain name reference, and such names must be acquired via some communication channel and cannot be guessed. For instance, the exit and enter operations allow a component to move up in the structure at the level of the current parent program and to move down in the structure inside the domain associated with one of its peers, respectively. In both cases, the correct name of the parent or the sibling must be known in order for the operation to succeed. This will become apparent only when we discuss the coordination semantics expressed in the **Interactions** section since both operations reduce simply to appropriate requests for action:

$x :=$**exit** n **if** $g \equiv$
$\quad \langle OP := (exit, n)$ **if** g; $x := true$ **if** $OP \uparrow 1 = pass$; $OP := nil \rangle$
$y :=$ **enter** n **if** $g \equiv$
$\quad \langle OP := (enter, n)$ **if** g; $y := true$ **if** $OP \uparrow 1 = pass$; $OP := nil \rangle$

The variables x and y are used to communicate back to the program that the operation succeeded. We use a transaction to set the variables x and y to the correct values after the coordination is completed.

In Mobile Ambients, **open** n dissolves the boundaries of a peer level ambient n. In our case, this is equivalent to bringing the subordinate programs to the same level as the parent. The domain does not disappear, but it does become empty. Locally, the operation is encoded again simply as a request which may or may not be satisfied:

$x :=$ **open** n **if** $g \equiv$
$\quad \langle OP := (open, n)$ **if** g; $x := true$ **if** $OP \uparrow 1 = pass; OP := nil \rangle$

The most subtle aspect of our attempt to create a structured navigation schema along the lines defined by Mobile Ambients is the management of domain names. In process algebras, the name restriction operator provides a powerful mechanism for generating new, unique names and for using them to enforce private communication among components. The operational approach associated with a programming notation such as Mobile UNITY forces us to consider an operational alternative that can offer comparable capabilities. Our solution is to permit domain (i.e., program) renaming. A renamed program cannot be referenced by anyone else unless the new unique name is communicated first—this is the analog of scope extension in process algebras.

The renaming operation assumes the form:

$d :=$ **rename** n **if** $g \equiv$
$\quad \langle OP := (rename, n, new())$ **if** $g; d := nil;$
$\quad\quad d := OP \uparrow 3$ **if** $OP \uparrow 1 = pass; OP := nil \rangle$

When renaming is successful, the new domain name is returned in d in order to facilitate it being communicated to other components. In principle, a component

may be able to rename itself, its domain (i.e., its parent), and its peers—as long as it has their correct names. This can be restricted further if necessary.

The **Interactions** section needs to encode the coordination rules associated with the operations above. The general pattern is to verify that the referenced name is correct, record in the variable OP this fact, and complete all necessary changes to the domain structure. We illustrate this by considering the case when a request is made to rename the current domain, i.e., the parent name:

$$P.OP := (pass, n, m) \;\; \textbf{reacts-to}\; P.OP = (rename, n, m) \wedge P.\lambda \uparrow 2 = n$$
$$Q.\lambda := (m, Q.\lambda \uparrow 2) \;\; \textbf{reacts-to}\; P.OP = (pass, n, m) \wedge Q.\lambda \uparrow 1 = n$$
$$R.\lambda := (R.\lambda \uparrow 1, m) \;\; \textbf{reacts-to}\; P.OP = (pass, n, m) \wedge R.\lambda \uparrow 2 = n$$

The first reactive statement records the success of the renaming for the case when n is indeed the domain name containing P, the initiator of the operation. The second reactive statement changes the domain name while the third changes the domain reference in all the components associated with the renamed domain. Similar code can be used to process exit, enter, and all other open requests.

As an illustration let us consider two programs P and Q which desire to share private information in a protected domain, and let us assume the existence of a third program S. Initially P, Q, and S are assumed to be part of some domain U, as shown in Figure 4a.
We use superscripts to denote the domain names. Assuming that P and Q know the name γ of S, they both can issue the operation **enter** γ changing the configuration to that shown in Figure 4b.

At this point, P can rename S with a new unique name δ and communicate the name δ to Q. The resulting configuration is shown in Figure 4c. Now, both P and Q can exit and enter S at will with no risk that any other program might be able to enter their private domain.

One problem this example ignores is the situation that some other program R may have entered S prior to P and Q. While R is trapped forever (R cannot perform any operations on S because the name of S is changed), R could interfere with the data exchanges between P and Q. There are several ways to avoid this situation. One interesting solution is to allow P to know the cardinality of its domain, i.e., the number of components in S.

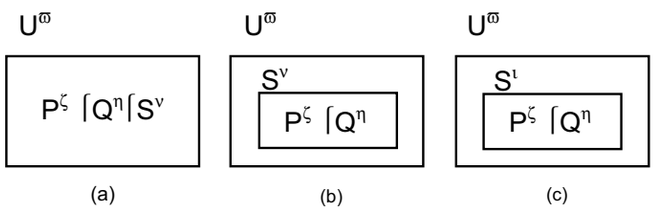


Fig. 4. Domain configurations

7 Location-Sensitive Synchronous Communication

The study of synchronous communication has its origins in CSP [8] and was later refined with the introduction of an entire family of process algebras, including CCS [13] and π-calculus [14]. Most often, events are identified by naming a communication channel and are differentiated as being send ($\overline{c}$) and receive (c) events associated with distinct processes. Pairs of matching events are executed simultaneously. If data is actually being exchanged, the typical notation is $c!x$ for sending a value and $c?x$ for receiving a value. In principle, many pairs of processes could be synchronized using the same channel name with matching pairs being selected nondeterministically. In π-calculus it becomes possible to protect access to a specific channel by creating new channel names and communicating them to other specific processes. In this section, we explore a schema that has these kinds of features and we show how communication can be constrained based on the relative location among processes. Let us consider first a generic event model, á la CSP [8], in which a process can send ($c!x$) or receive ($c?x$) values on a specified channel. Since Mobile UNITY programs are not sequential in nature, blocking will be interpreted as no additional operations being permitted to take place on the respective channel. Finally, we allow a process to use the same channel in both directions, but not at the same time.

Under these assumptions, an output operation may assume the following syntax and semantics:

$c!x$ **if** $g \equiv$
$\quad c := (out, x)$ **if** $g \wedge c = nil$

The local view of the channel stores a request for output, if not already in use. If the guard is passable and the channel is not in use, a request for an input operation works similarly, but requires the transfer of the channel value to a specified variable:

$c?x$ **if** $g \equiv$
$\quad c := (in, nil)$ **if** $g \wedge c = nil \ [] \ x, c := c \uparrow 2, nil$ **reacts-to** $c \uparrow 2 \neq nil$

The reaction guarantees the immediate data transfer and the resetting of the channel state.

As before, the actual coordination takes place in the **Interactions** section. The standard solution is to simply match pairs of pending input/output operations present in different processes and involving the same channel:

$P.c, Q.c := nil, (in, P.c \uparrow 2)$ **when** $P.c \uparrow 1 = out \wedge Q.c \uparrow 1 = in$

where process names P and Q and channel name c are universally quantified.

Another alternative is to carry out communications as soon as they are feasible. This can be accomplished simply by replacing the asynchronous transfer above by a corresponding reactive statement:

$P.c, Q.c := nil, (in, P.c \uparrow 2)$ **reacts-to** $P.c \uparrow 1 = out \wedge Q.c \uparrow 2 = in$

The final variation on this theme is to restrict such communications to situations in which P and Q are co-located, as in:

$P.c, Q.c := nil, (in, P.c \uparrow 2)$
reacts-to $P.c \uparrow 1 = out \wedge Q.c \uparrow 2 = in \wedge P.\lambda = Q.\lambda$.

So far we assumed that the channel names were fixed and the lack of any protection against unauthorized usage. In order to capture the unique ability of π-calculus to create new channel names that can be passed around among processes, we need to distinguish between the variable used to refer to a channel and the channel name. By storing the channel name, it becomes possible for it to be changed and shared. Surprisingly, the changes in the schema are relatively straightforward. First, we assume the existence of a function that returns a unique system-wide name that can be stored in a local program variable and cannot be forged:

$\eta := new()$.

Second, we alter the structure of the local channel to accept a new name, but only when not in use:

c **named** η **if** $g \equiv$
$c := (\eta, nil)$ **if** $g \wedge c \uparrow 2 = nil$

The send and receive operations are altered so as to not impact the channel name. The requests are stored in the second field associated with the local view of the channel:

$c!x$ **if** $g \equiv$
$c := (c \uparrow 1, (out, x))$ **if** $g \wedge c \uparrow 2 = nil$
$c?x$ **if** $g \equiv$
$c := (c \uparrow 1, (in, nil))$ **if** $g \wedge c \uparrow 2 = nil$
$[\!]$ $x, c := c \uparrow 2 \uparrow 2, (c \uparrow 1, nil)$ **reacts-to** $c \uparrow 2 \uparrow 2 \neq nil$

Finally, input/output commands are matched based on names associated with the individual channels, and, if desired, constrained to co-location:

$P.a, Q.b := (P.a \uparrow 1, nil), (Q.b \uparrow 1, (in, P.a \uparrow 2 \uparrow 2))$
where $P.a \uparrow 1 = Q.b \uparrow 1 \wedge P.a \uparrow 2 \uparrow 1 = out$
$\wedge\ Q.b \uparrow 2 \uparrow 1 = in \wedge P.\lambda = Q.\lambda$.

The result is an interesting combination of a mobility schema with a dynamic reconfiguration schema.

8 Conclusion

The premise of this position paper has been the notion that effective use of powerful models in practical settings must rely on disciplined use of the model. One formal strategy for establishing such a discipline of thought and design is to impose an appropriate programming schema, i.e., mostly syntactic restrictions over the model. We explored the schema-based approach in the particular domain of coordination models and languages for mobility and produced evidence that Mobile UNITY can be readily customized to accommodate a diverse set of coordination models in a manner that maintains the clean separation between fully decoupled component actions and the coordination semantics that tie them together. We view the examples we provided mostly as exercises designed to illustrate ideas rather than solve specific problems or define novel formal strategies. Nevertheless, some of the illustrations offer practical guidelines for the use of a formal notation such as Mobile UNITY as the basis for precise definitions of coordination-based middleware constructs. At the same time, some of the exercises suggest interesting new investigative paths deserving of a more careful formal analysis.

Acknowledgements

This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2–3):133–180, 1990. 128

[2] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000. 127, 138

[3] L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science, Special Issue on Coordination*, 240(1):177–213, June 2000. 127, 143

[4] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, New York, NY, 1988. 126, 134

[5] N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: A tuple space based platform for adaptive mobile applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, May 1997. 138

[6] D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985. 138

[7] R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. 1998. 127, 138

[8] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. 147

[9] C. Mascolo. MobiS: A specification language for mobile systems. In *Proceedings of 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 37–52. Springer-Verlag, 1999. 143

[10] C. Mascolo, G. P. Picco, and G.-C. Roman. A fine-grained model for code mobility. In *Proceedings of the Seveth European Software Engineering Conference ESEC*, volume 1687 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, September 1999. 143

[11] P. J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998. 126, 127, 128, 134

[12] P. J. McCann and G.-C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 8(2):115–146, April 1999. 127

[13] R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989. 147

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992. 127, 147

[15] A. L. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. PhD thesis, Washington University in St. Louis, August 2000. 143

[16] A. L. Murphy, G.P Picco, and G.-C. Roman. LIME: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Systems*, pages 524–533. IEEE Computer Society Press, April 2001. 127, 142

[17] A. Omicini and F. Zambonelli. The TuCSoN coordination model for mobile information agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, June 1998. 138

[18] G. P. Picco, G.-C. Roman, and P. J. McCann. Reasoning about code mobility in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 10(3):338–395, 2001. 127

[19] G.-C. Roman and P. J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20:47–68, 2002. 126, 127, 128

UML and Concurrency

Perdita Stevens

School of Informatics, University of Edinburgh
perdita@inf.ed.ac.uk.
Fax: +44 131 667 7209

Abstract. In this paper we survey the various ways in which concurrency concerns interact with the modelling of software systems in UML. We discuss the aspects of UML1.4 which can be used to model concurrency aspects of systems, and mention some of the changes expected in UML2.0. We briefly survey some of the research which has been carried out concerning UML and concurrency, and some outstanding issues.

1 Introduction

The Unified Modelling Language has been widely adopted as a standard language for modelling the design of (software) systems. It origins were primarily in the modelling of what we may call mainstream business applications, where real-time constraints were few and concurrency was rarely an issue. The success of UML, however, has seen it applied far more widely; at the same time, the continued rise of distribution and the increasing importance of the Internet has meant that concurrency concerns more UML users than previously.

The aim of this paper is to serve as an introduction to concurrency issues in UML which may be a useful starting point for people wishing to get into the area. The paper is structured as follows. We start by considering the ways in which concurrency appears in the UML standard, and in the latest draft of the forthcoming standard. Next we consider some of the research work which has been done concerning concurrency issues in UML, with apologies to any authors whose work has been inadvertently omitted. Finally we discuss open issues.

1.1 Note on Standards and Terminology

At the time of writing (October 2002) the current UML standard is version 1.4 [19]. It is expected that by the time of the ASM'03 workshop, the final proposal for UML 2.0, which is a significant revision of the language, will have been presented to the OMG. This paper is based on UML1.4 and on the latest draft at the time of writing of the main consortium's UML2.0 superstructure proposal [20].

When considering research which has been done on UML and concurrency it is sometimes important to be aware of what version of UML is being used: the language has changed in many important respects over its lifetime. Authors are not always explicit about this.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 151–166, 2003.
© Springer-Verlag Berlin Heidelberg 2003

2 What Do We Mean by Concurrency?

In UML, "concurrency" is often conflated with real-time and/or distributed systems design.

In theoretical computer science, on the other hand, "concurrency" is used as a general term for a field of work which is often held to include, for example:

- all work on process algebras, even process algebras (such as BPA) which do not have a parallel operator;
- all work on temporal logics
- much work on real-time and distributed systems.

Part of what's going on here is that concurrency and non-determinism occur so often together that they are confused. The cause of non-determinism in systems is often that two or more components are acting simultaneously and their actions may occur in different orders from the point of view of an observer. On the other hand, non-determinism may also occur in a specification or design model because certain decisions have not yet been made; and the presence of an outside user with its own thread of control, such as a human, will cause the effect even though we would not generally wish to regard all systems with human users as concurrent.

In this paper we will try to take a relatively purist approach: an issue concerns concurrency if it concerns modelling two or more things happening in parallel.

3 Concurrency in the UML Standard

We will consider the different diagram types in turn, discussing features relevant to concurrency in each case. In some cases we raise issues with the concurrency features, but we do not cite any research work which addresses these issues: that is in the next section.

3.1 Use Case Diagrams

A use case diagram gives an overview of the functionality of the system as perceived by its users and by systems external to it (collectively, *actors*). The different *use cases* correspond to tasks which might be carried out with the help of the system. A *scenario* is the execution of an instance of a use case: that is, a sequence of messages exchanged between the system and the actors. The diagram does not record what constraints there may be on the order in which use cases are carried out or the combinations of use case instances which might be active at one time: use cases are generally thought of as though they were independent (except in the case of ≪include≫ and ≪extend≫ relationships). Indeed [19] 2-141 says "The service, which is initiated by a user, is a complete sequence. This implies that after its performance the entity will in general be in a state in which the sequence can be initiated again."; this assumes a certain level of independence.

This is a natural and generally useful abstraction, but it can only be an approximation to the truth. Use cases are not parts of the system, so they do not communicate; instead, each use case specifies an interaction between the system being modelled and external systems and users. Eventually, when the system is implemented, there will be a possibility that the fact that one scenario is being carried out will interfere with the execution of another, simply because both affect the state of the system (and possibly of the actors). In practice, it may not matter if these issues are neglected, since use cases are typically used informally. It may be more significant, however, for formal or tool-oriented work.

3.2 Class Diagrams

Concurrency concerns generally do not appear in class diagrams. The exception is that there is a concurrency attribute of Operation. Possible values of this attribute are sequential, guarded, and concurrent; the possibility is left open of defining other possibilities. The descriptions in [19] 2-88 of these three options are as follows:

sequential Callers must coordinate so that only one call to an Instance (on any sequential Operation) may be outstanding at once. If simultaneous calls occur, then the semantics and integrity of the system cannot be guaranteed.

guarded Multiple calls from concurrent threads may occur simultaneously to one Instance (on any guarded Operation), but only one is allowed to commence. The others are blocked until the performance of the first Operation is complete. It is the responsibility of the system designer to ensure that deadlocks do not occur due to simultaneous blocks. Guarded Operations must perform correctly (or block themselves) in the case of a simultaneous sequential Operation or guarded semantics cannot be claimed.

concurrent Multiple calls from concurrent threads may occur simultaneously to one Instance (on any concurrent Operation). All of them may proceed concurrently with correct semantics. Concurrent Operations must perform correctly in the case of a simultaneous sequential or guarded Operation or concurrent semantics cannot be claimed.

3.3 Collaboration and Sequence Diagrams

Collaboration diagrams and sequence diagrams are both kinds of interaction diagrams, and they are essentially interchangeable: many tools, indeed, offer automatic conversion between the diagram types. Figure 1 shows a sequence diagram and Figure 2 shows a collaboration diagram for the same interaction, which includes some features which we will discuss.

The main features relevant to concurrency are active objects and concurrent messages.

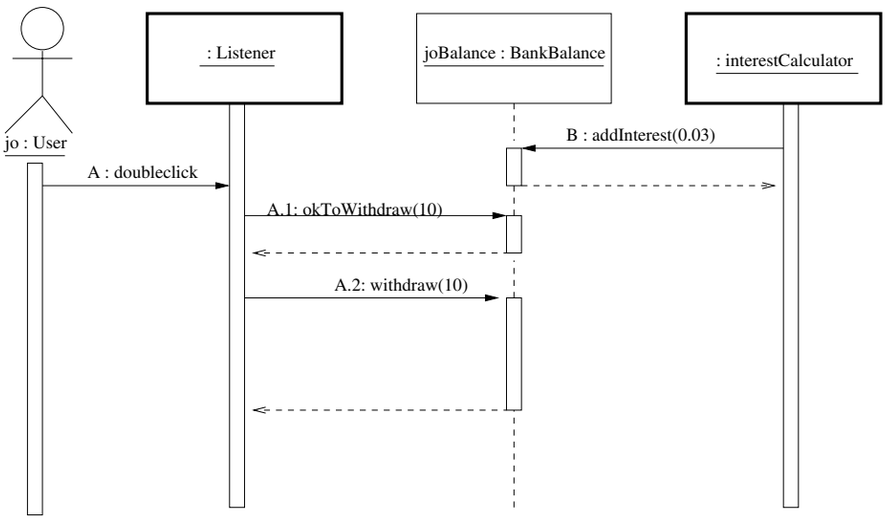


Fig. 1. Sequence diagram

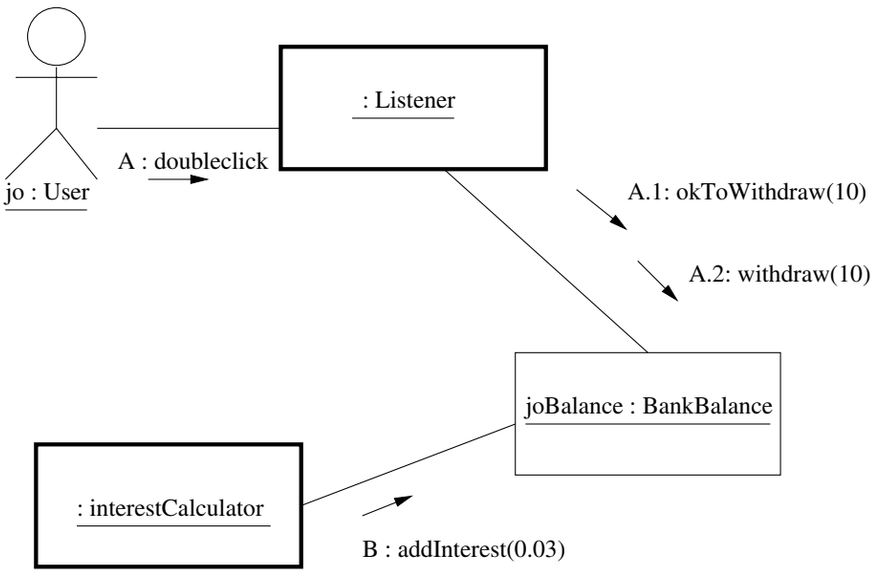


Fig. 2. Collaboration diagram

Active objects and classes The Notation Guide section of UML1.4 describes an "active object" as one which "owns a thread of control and may initiate control activity." For example, a thread or a process is expected to be modelled, if at all, as an active object. An active object is represented like an ordinary object, by a rectangle, but with a thick border. A collaboration diagram which makes use of the notation to distinguish active from passive objects is sometimes called a *concurrent collaboration diagram*.

The active object concept is reflected in the UML metamodel by the attribute "isActive" of metaclass Class, which "Specifies whether an Object of the Class maintains its own thread of control. If true, then an Object has its own thread of control and runs concurrently with other active Objects."

Interestingly, despite the fact that the UML semantics identifies classes, not objects, as being active, the Notation Guide does not discuss or suggest a notation for an "active class". The expectation seems to be that active objects are used in collaboration diagrams, which represent the interactions which take place between the different objects of the system, but that there is no need to specially identify the classes of those objects.

Concurrent messages Messages in an interaction diagram are typically numbered. In a sequential system, the numbering scheme uses a dot-separated structured format to record the nesting of calls. For example, if an object o receives a message m labelled 1.2, then the first message which o sends as parts of its reaction to receiving m will be labelled 1.2.1.

Where an interaction involves the sending of several messages in parallel, letters rather than numbers can be used. For example, a diagram might involve messages 1.A, 1.B, 1.A.1 etc. to show that two messages are sent in parallel in response to message 1. It might be useful to use this notation also in cases where the messages will be sent in some definite order in the implemented system, but that order has not yet been decided. However, the UML standard does not suggest that this is legitimate.

Where a collection of similar messages is sent in parallel, a parallel iteration can be used. For example, a message labelled $*\|[i := 1..n]$ will be sent n times in parallel.

This notation may be useful in interaction diagrams that involve a very small amount of parallelism. However, the notation is rather clumsy and inflexible; one would not want to attempt to use it for complex concurrent behaviour.

Within the UML semantics, a rather more flexible mechanism is provided by explicit identification of the predecessors of a given message. A message may have more than one predecessor, for example when two threads synchronise, or two messages may have the same predecessor, for example when a new thread forks. However, there is no graphical notation allowing the mechanism to be used in full generality on a diagram; for example, there is no defined notation for synchronisation.

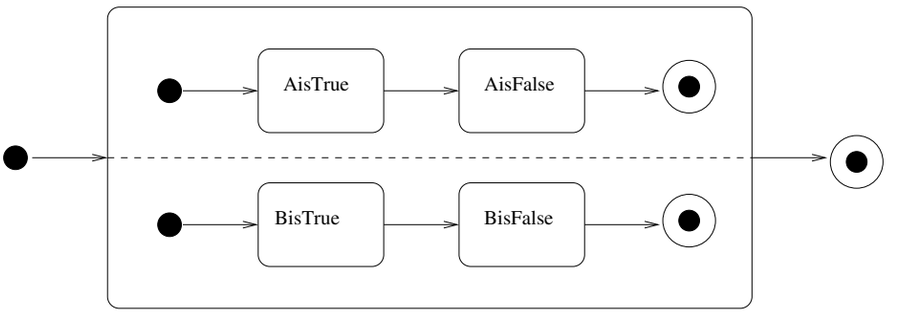
Notice that the sequence diagram illustrated visually incorporates the assumption that both A.1 and A.2 happen after B; no such assumption is in-

corporated in the collaboration diagram. In the underlying model, this will be disambiguated by the predecessor relation.

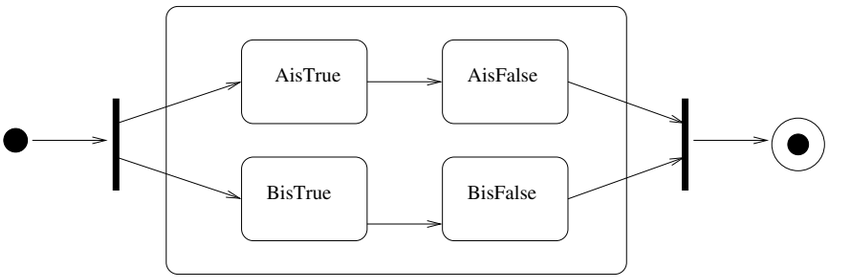
3.4 State and Activity Diagrams

State diagrams are generally used to represent the possible abstract states of a classifier, such as a class, and how an instance of the classifier moves between those states when it reacts to events, such as receiving a message. Activity diagrams are more often used to represent the steps involved in fulfilling a use case or carrying out an operation. We consider them together because up to and including UML1.4, activity diagrams have been treated as a special case of state diagrams.

This is expected to change in UML2.0; the proposal [20] gives activity diagrams a semantics based on petri nets, whereas state diagrams retain essentially their existing semantics.



(a) Explicitly shown concurrent regions



(b) The same semantics, using forks and joins instead

Fig. 3. State diagram

Concurrent substates in state diagrams Figure 3 shows two versions of a UML state diagram with two concurrent substates. To specify an object's state we need to specify its substate in both of these concurrent substates. In (a) the concurrent substates are shown explicitly, separated by a dashed line. In (b) they are implicit, and instead the splitting of the state into concurrent regions is shown using a fork (the left hand heavy bar) and subsequent join (the right hand heavy bar).

One practical concern which does not seem to be addressed explicitly in the UML standard is what concurrent regions actually mean for an implementation: for example, in an active object, must they represent states of different threads? Conversely, is it legitimate to use concurrent substates just to represent orthogonality in the state space? For example, if a passive, sequential object has two boolean attributes which can be changed independently, is it legitimate to represent its state chart as in Figure 3? Or does the notation imply "real" concurrency?

Run-to-completion and concurrency UML's semantics for state diagrams is *run-to-completion*; essentially this means that a step in the semantics begins with the dispatch of an event, which causes one or more transitions to fire; any actions caused by the firing of the transitions must also be complete before the step in the semantics is deemed to be complete. This raises some issues for semantics of concurrent regions. UML1.4 says:

> It is possible to define state machine semantics by allowing the run-to-completion steps to be applied concurrently to the orthogonal regions of a composite state, rather than to the whole state machine. This would allow the event serialization constraint to be relaxed. However, such semantics are quite subtle and difficult to implement. Therefore, the dynamic semantics defined in this document are based on the premise that a single run-to-completion step applies to the entire state machine and includes the concurrent steps taken by concurrent regions in the active state configuration. In case of active objects, where each object has its own thread of execution, it is very important to clearly distinguish the notion of run to completion from the concept of thread pre-emption. Namely, run-to-completion event handling is performed by a thread that, in principle, can be pre-empted and its execution suspended in favor of another thread executing on the same processing node. (This is determined by the scheduling policy of the underlying thread environment no assumptions are made about this policy.) When the suspended thread is assigned processor time again, it resumes its event processing from the point of pre-emption, and eventually completes its event processing.

Semantics of concurrent state machines have been addressed by a number of researchers, addressed later.

Synch states SynchStates are used for the synchronisation of concurrent regions. Technically they are not states at all; for those familiar with petri nets, a state

is by definition a place which may only contain one token (present if the state is active, absent otherwise) whereas a synch state is a place which may be given explicitly an upper bound on the number of tokens it may contain. If further tokens arrive (by means of the firing of an incoming transition), they are discarded; firing an outgoing transition results in the loss of one token. The idea is that transitions that enter a synch state are coming from forks in different concurrent regions, and transitions that leave a synch state are going to joins. Thus the join cannot be reached until after the fork has been reached, forcing the concurrent region where the join is to wait for the region where the fork is. The join cannot be reached more times than the fork has been reached, because the synch state will run out of tokens. The bound on the synch state enforces a maximum number of times that the join can be reached between occasions when the fork is reached. A bound of $*$ may be, and often is, used to indicate that the synch state is unbounded.

Fork and join notation in state and activity diagrams A simple activity diagram is shown in Figure 4. It shows a fork (top horizontal bar) followed by a join (bottom horizontal bar). The two activities in between are carried out in parallel, or perhaps in a non-deterministic interleaving: as usual, UML does not distinguish.

On a simple level it is easy to see activity diagrams as a notational variant of Petri nets, and this is exploited in the UML2.0 proposal mentioned earlier. Currently, however, UML's intention seems to be that forks and joins are a merely diagrammatic alternative to concurrent states in a state machine: the UML specification ([19]2-183) includes a well-formedness rule which says

> All of the paths leaving a fork must eventually merge in a subsequent join in the model. Furthermore, multiple layers of forks and joins must be well nested, with the exception of forks and joins leading to or from synch state. Therefore the concurrency structure of an activity graph is in fact equally restrictive as that of an ordinary state machine, even though the composite states need not be explicit.

However, it is notable that this rule is only expressed in English; the usual attempt to make rules precise using OCL is not followed, and indeed the corresponding well-formedness rules concerning forks and joins in state diagrams ([19] 2-157, 2-158) explicitly exclude activity diagrams from their scope.

An argument for relaxing this rule, and allowing an alternative semantics of activity diagrams, is that the rule would seem to exclude some perfectly sensible diagrams. For example, consider Figure 5, adapted from [2]. Because of the "cross-synchronisation", it cannot be redrawn as a state diagram with concurrent regions; however, it could reasonably represent the dependencies between activities making up an operation or a use case.

Nevertheless we may well want to exclude some of the diagrams we can draw with the available notation. For example, what if anything should the diagram in Figure 6 mean?

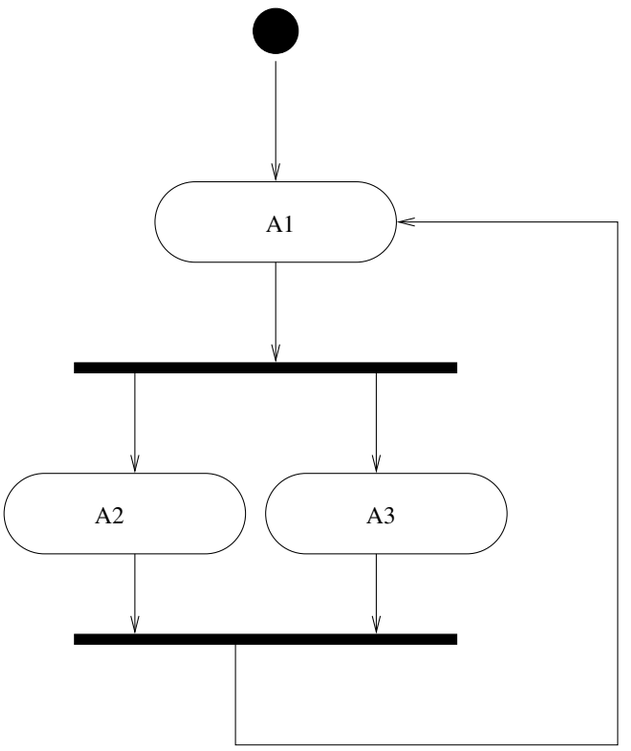


Fig. 4. A simple activity diagram

It is to be hoped that founding activity diagrams on petri nets opens the door to a more precise semantics that allows flexible use of the notation. However, I have not yet studied the treatment described in the draft proposal in detail.

3.5 OCL

The Object Constraint Language provides a way of specifying properties of a particular instance in the execution of a system implementing a UML model. Thus it does not concern itself with concurrency; in particular, it has no temporal operators, although several researchers have proposed ways of extending OCL.

4 UML Research Involving Concurrency

The workshop on Concurrency Issues in UML [13] at UML2001 was introduced to its potential participants by means of a white paper [14] outlining UML's concurrency mechanisms (much as above, but with the emphasis on UML as

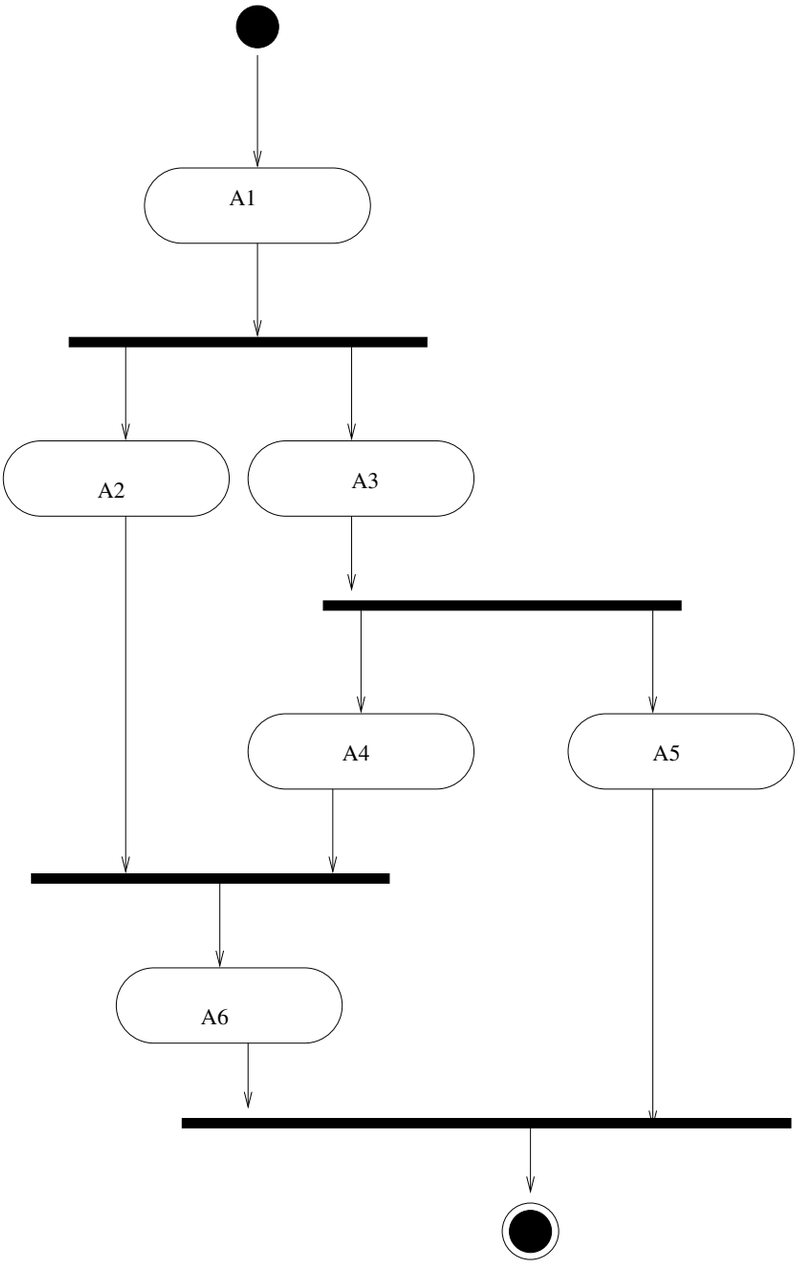


Fig. 5. A reasonable, but probably illegal activity diagram

a programming language) and raising a number of issues where clarification or further work was needed. Only some of the papers from that workshop can be cited here, but all are currently available on line.

4.1 Use Case Diagrams

Work from a concurrency perspective includes the author's [23]. In this paper it was argued that use cases can sensibly be modelled by labelled transition systems, and in particular that the representation of choice points is essential: despite the UML standard's occasional suggestion that a use case is synonymous with the *set* of scenarios (traces) which comprise it, this is not defensible. However, this paper only considered the modelling of one use case: it did not consider what happens when several scenarios from the same or different use cases execute simultaneously.

4.2 Class Diagrams

Newman, Greenhouse and Scherlis [17] identify that in shared data concurrent applications it is important to design carefully what data is shared between

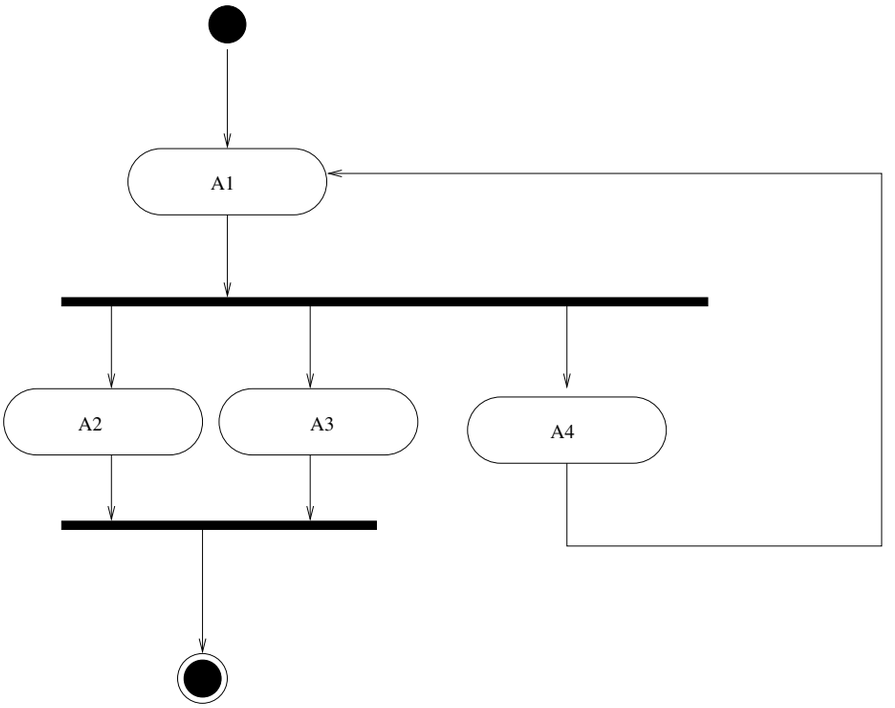


Fig. 6. An arguably unreasonable activity diagram

which participants and how it is protected. They propose an extension of class diagrams called *regional state hierarchy diagrams* to represent shared state and its protection. (They also propose *method concurrency diagrams* but these are presented as a new diagram type, not as a variant of any existing UML features.)

4.3 Interaction Diagrams

Mehner and Wagner [16] suggested a mechanism involving new stereotypes to allow the synchronisation of Java threads to be represented visually in collaboration diagrams; they also make use of *temporary active objects*.

4.4 State Machines

An area where members of the ASM community have been particularly active is semantics of state machines [3, 5, 6] and in this volume [18, 8]. In particular Börger, Cavarra and Riccobene provide a semantics that allows different resolutions of UML's "semantic variation points", and draws attention to some anomalies in the treatment of conflicts between transitions originating in different concurrent regions. There is also a wide variety of other approaches to the semantics of state diagrams, of which I omit discussion here.

4.5 Activity Diagrams

Within the ASM community, [4] provided an ASM semantics for activity diagrams. A process algebraic approach was taken by [2]; in particular, this approach permitted the use of activity diagrams with cross-synchronisation as shown in Figure 5.

Eshuis and Wieringa have worked extensively in the area of formal semantics for activity diagrams; see for example [11, 10]. They do not always adhere to the UML semantics; their main concern is the use of activity diagrams for workflow. In [12] there is a discussion of Petri nets as a semantics for activity diagrams.

Petriu and Wong [21] propose a style of using activity diagrams with explicit sending and receiving of signals to model concurrent operations in a message-passing style; a concern of theirs is to provide appropriate places for performance annotations.

4.6 Methodology

There are also several strands of work in which authors propose development methods for concurrent systems. These authors include Sendall and Strohmeier (in for example [22], one of whose main contributions is an extension of OCL to allow *operation schemas* to specify synchronisation on shared resources) and Gomaa [15].

4.7 Work on Related Notations

In addition to work which is identified by the authors as being related to UML there is also a great deal of work which relates to the notations which preceded UML, such as Harel statecharts and message sequence charts. In the MSC community there has been interesting work on treating sets of MSCs together as specifications of behaviour, and asking questions such as "is it possible to implement a system which satisfies all of these MSCs?" (realizability) and "if a system realises all of these MSCs, what others must it automatically realise?" (inference) [1]. Another aspect is to use *live sequence charts* (LSCs) [9], which allow the designer to distinguish between behaviour which may happen and behaviour which must happen. Harel and collaborators have recently been developing a "play-in, play-out" approach to exploiting the usability of sequence diagram-like notations, implemented in a tool called the Play Engine[1].

5 Issues

In this section we consider a few more general issues of concurrency and UML.

Interleaving or "true concurrent" semantics? Generally, UML semantics work concerned with concurrency maps a UML model to a labelled transition system (or to some other formal object which is itself mapped to an LTS). That is, interleaving semantics is used: the mathematical model does not model events happening "truly" concurrently, but rather reflects the idea that they can happen in either order. Concurrency is mapped to non-determinism. A major advantage of this approach is that verification based on such semantics is well-established. However, there are also some disadvantages. If events really happen instantaneously – which is in most modelling contexts assumed – then the interleaving model does not violate most people's intuitions too badly. If the time taken by an event becomes significant, one starts to want a model in which it is possible to model events overlapping. Another consideration is that interleaving models suffer from state explosion as a consequence of mapping concurrency to non-determinism. For example, if five events happen simultaneously, an interleaving model will record paths for each of the 5! possible orders of the events, involving $2^5 = 32$ states. In the verification community many techniques are known for mitigating this state explosion problem; but it is natural to wonder whether the whole problem could be side-stepped by using a different model. However, verification using true concurrent models is in its infancy, and the results so far are not encouraging, with basic problems proving undecidable. There is an outstanding challenge to develop natural, intuitive, and yet tractable models suitable for verification of concurrent systems modelled in UML.

Refinement, instantiation and generalisation There are many circumstances in which UML diagrams should be related by some kind of refinement relationship. For example:

[1] http://www.wisdom.weizmann.ac.il/~rami/PlayEngine/

- we might be carrying out a traditional refinement step, gradually making more decisions about how a system is to be implemented
- we may develop one diagram for a class and want to compare it with a diagram for a subclass
- we may develop a diagram to specify a framework or product-line architecture, and want to check that a diagram specifying a particular instantiation of the framework or PLA is consistent with it.

Some such situations have been considered within the UML community, but I have seen little appreciation of the vast choice of different refinement-like relations that have been studied in the concurrency community. (In [23] I briefly addressed the choice for generalisation of use cases.) Choosing the appropriate relation is a difficult and interesting problem; in particular, there is not likely to be one right answer that suffices for all situations.

Working with partial information In the concurrency community we can often assume that we are being given a model which, at some level of abstraction, carries complete information about the system. This assumption also holds for some uses of UML, particularly among real-time and embedded system developers where UML tends to be used more as a graphical programming language than as a design modelling language. However, in most UML models the information which the designer chooses to record is very incomplete. For example, only some classes will be given state diagrams; the state diagrams may record some parts of objects' reactions to messages, but not the full detail; only a few interaction diagrams may be provided; and so on.

Clearly for most verification tasks, full information is required. But the realities and economics of software development mean that we are never likely to be able to assume that all software developers will always provide fully detailed models. What can we do to help in any case? This is partly a challenge for tools, which have to be able to help the user to extend models easily, and partly a challenge for theoreticians, who have to be able to reason with partial information.

Acknowledgements

I am grateful to the Programme Chairs of the ASM'03 workshop for the invitation that led to this paper, and to the EPSRC funding in the form of an Advanced Research Fellowship.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Inference of message sequence charts. In *Proc. International Conference on Software Engineering*, 2000. 163

[2] Christie Bolton and Jim Davies. Activity graphs and processes. In *Proc. IFM 2000*, number 1945 in LNCS. Springer, 2000. 158, 162

[3] E. Börger, A. Cavarra, and E. Riccobene. Solving conflicts in UML state machines concurrent states. In [13]. 162

[4] E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In *Proc. AMAST 2000*, volume 1826 of *LNCS*. Springer, 2000. 162

[5] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML state machines: making semantic variation points and ambiguities explicit. In *Proc. of Semantic Foundations of Engineering Design Languages (SFEDL), Satellite Workshop of ETAPS 2002*, 2002. 162

[6] E. Börger, A. Cavarra, and E. Riccobene. Modeling the meaning of transitions from and to concurrent states in UML state machines. In *ACM SAC 2003 Software Engineering Track*, 2003. 162

[7] E. Börger and Elvinia Riccobene, editors. *10th International Workshop on Abstract State Machines*, LNCS. Springer, 2003. (this volume). 165

[8] A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines. In *[7]*, 2003. 162

[9] Werner Damm and David Harel. LSCs: breathing life into message sequence charts. *Formal methods in systems design*, 19(1):45–80, 2001. 163

[10] Rik Eshuis and Roel Wieringa. An execution algorithm for UML activity graphs. In *Proc. UML 2001*, volume 2185 of *LNCS*. Springer, 2001. 162

[11] Rik Eshuis and Roel Wieringa. A real-time execution semantics for UML activity diagrams. In *Proc. Fundamental Approaches to Software Engineering*, volume 2029 of *LNCS*, pages 76–90. Springer, 2001. 162

[12] Rik Eshuis and Roel Wieringa. Comparing petri nets and activity diagram variants for workflow modelling- a quest for reactive petri nets. In *Petri Net Technology for Communication Based Systems (to appear)*, LNCS. Springer, 2003. 162

[13] C. Atkinson et al., editor. *Workshop on Concurrency Issues in UML*, 2001. proceedings unpublished; available from `http://wooddes.intranet.gr/uml2001/Home.htm`. 159, 164, 165

[14] Sébastien Gérard and Ileana Ober. Parallelism/concurrency specification within UML. White paper connected with [13]. 159

[15] Hassan Gomaa. *Designing concurrent, distributed and real-time applications uwith UML*. Addison-Wesley, 2000. 162

[16] Katharina Mehner and Annika Wagner. Visualizing the synchronization of Java-threads with UML. In *Proceedings of Visual Languages 2000*, 2000. 162

[17] Elissa Newman, Aaron Greenhouse, and William L. Scherlis. Annotation-based diagrams for shared data concurrency. In [13]. 161

[18] Ileana Ober. An ASM semantics of UML derived from the meta-model and incorporating actions. In *[7]*, 2003. 162

[19] OMG. *Unified Modeling Language Specification version 1.4*, September 2001. OMG document formal/01-09-67 available from http://www.omg.org/technology/documents/formal/uml.htm. 151, 152, 153, 158

[20] U2 Partners. Unified Modeling Language 2.0 proposal, version 2 beta R1, September 2002. 151, 156

[21] Dorina C. Petriu and Eric Wong. Using activity diagrams for representing concurrent behaviour. In [13]. 162

[22] Shane Sendall and Alfred Strohmeier. Specifying concurrent system behaviour ant timing constraints using OCL and UML. In *Proc. UML 2001*, number 2185 in LNCS. Springer, 2001. 162

[23] Perdita Stevens. On use cases and their relationships in the Unified Modelling Language. In *Proc. Fundamental Approaches to Software Engineering*, number 2029 in LNCS, pages 140–155. Springer-Verlag, April 2001. 161, 164

A Unified Formal Specification and Analysis of the New Java Memory Models

Varsha Awhad and Charles Wallace

Computer Science Dept., Michigan Technological University
Houghton, Michigan, USA

Abstract. In a multithreaded program running on a multiprocessor platform, different processors may observe operations in different orders. This may lead to surprising results not anticipated by the programmer. The problem is exacerbated by common compiler and hardware optimization techniques. A memory (consistency) model provides a contract between the system designer and the software designer that constrains the order in which operations are observed. Every memory model strikes some balance between strictness (simplifying program behavior) and laxness (permitting greater optimization).
With its emphasis on cross-platform compatibility, the Java programming language needs a memory model that is satisfactory to language users and implementors. Everyone in the Java community must be able to understand the Java memory model and its ramifications. The description of the original Java memory model suffered from ambiguity and opaqueness, and attempts to interpret it revealed serious deficiencies. Two memory models have been proposed as replacements. Unfortunately, these two new models are described at different levels of abstraction and are represented in different formats, making it difficult to compare them. In this paper we formalize these models and develop a unified representation of them, using Abstract State Machines. Using our formal specifications, we relate the new Java memory models to the Location Consistency memory model and to each other.

1 Introduction

As parallelism becomes ever more prevalent in the computing world, the *shared memory* paradigm is emerging as a popular approach to building parallel systems and applications. A *shared memory multiprocessor machine* is characterized by a collection of processors that exchange information with one another through a *global address space* [1]. In such machines, memory may be partitioned in many ways. For instance, in bus-based shared memory, processors access a separate monolithic main memory area over a bus; in distributed shared memory, each processor may have local access to some portion of the shared memory, but remote memory accesses require communication via a network. The advantage of shared memory is that programmers writing applications for such machines

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 166–185, 2003.
© Springer-Verlag Berlin Heidelberg 2003

are shielded from details about the physical distribution of memory. Each processor views the same global address space and accesses remote memory locations through the standard read and write instructions found in uniprocessor machines, rather than through special message-passing operations [13].

The behavior of a program running on a shared memory machine depends on the order in which memory operations are made visible to processors. Due to the physical distribution of memory, operations cannot be made visible instantaneously to all processors, so there is no simple notion of a single order of operations. Indeed, many common optimization techniques may complicate the ordering even further. Shared memory machines have various buffers where data written by a processor can be stored before they are shared with other processors. Freshly written values may remain in write buffers until it is convenient to propagate them to other processors. Processors may choose to read outdated values from their local caches to avoid the cost of remote reads. Furthermore, compilers may permute program instructions into an order more advantageous from a performance perspective. Similarly, processors may execute instructions out of program order. Typically, such reorderings are chosen carefully so as to be imperceptible in a uniprocessor context, but they cannot be hidden from other processors operating concurrently on a shared address space.

In a totally unregulated shared memory machine, programmers would find it impossible to predict the behavior of their applications. Optimizations at the hardware or compiler level can have consequences for even simple programs. To avoid complete chaos, every shared memory machine has an underlying *memory (consistency) model*, a contract between a program and the underlying machine architecture that constrains the order in which memory operations become visible to processors [1]. By constraining the order of operations, a memory model determines which values can legally be returned by each read operation. The most common memory model, *sequential consistency* (SC) [19], ensures that memory operations performed by the various processors are *serialized*, or seen in the same order by all processors, and that the common order preserves the order dictated by each program. This results in a model similar to the simple and familiar uniprocessor model.

Because of the requirement that all memory operations be serialized, the SC model is quite restrictive; it requires a high level of interprocessor communication in order to keep caches consistent, and it prohibits most reordering operations at the compiler and hardware levels. Other memory models [11, 8, 4, 25, 7] have been proposed to relax the requirements imposed by SC and therefore allow a wider range of optimizations. Relaxed memory models place fewer constraints on the memory system than SC, offering the promise of greater parallelism and less interprocessor communication. However, the results of some program executions may be quite counterintuitive to programmers familiar with uniprocessor or sequential consistency. As program behavior becomes less restricted, reasoning about programs becomes more difficult.

Memory models are subtle, hard to describe, and even harder for the average programmer to understand. The original memory model for the Java pro-

gramming language [12], written in natural language, suffered from serious imprecision and ambiguity [14]. The possibility of multiple interpretations of the memory model threatens Java's promise of "write once, run anywhere" software. Furthermore, the language designers themselves were victims of their memory model's complexity; they did not realize that their memory model not only disallowed many common optimizations but also failed to guarantee some basic safety properties [22].

There have been two recent attempts to come up with a better model: one by Manson and Pugh [21] and another by Maessen, Arvind and Shen [20] that uses the basic operations of the *Commit-Reconcile-Fence (CRF)* model [25]. We shall call the former the *Manson-Pugh JMM (Java memory model)* and the latter the *CRF JMM*. both models, the authors realize the importance of precise, formal description. However, the notations describing the two models are quite different. Manson and Pugh use an operational pseudocode reminiscent of ASM, while the CRF JMM is described through a combination of algebraic rules (imposing constraints on instruction reordering) and term-rewriting rules (giving the behavior of the memory system). Furthermore, the levels of abstraction of the two models are quite different. Manson and Pugh conceive of a history of abstract write operations whose relative ordering determines which values are readable at any given time. Where the values of these abstract writes are stored is intentionally left unspecified. The CRF JMM takes a lower-level view, describing a particular memory system architecture with a cache for each thread and a single common main memory area. Furthermore, Java operations are actually translated into finer-grained CRF operations, and rules are given for legal reorderings of CRF operations. The lower level of abstraction may be more satisfying in its concreteness, but it comes at a cost. For instance, in *cache-only memory architectures* [15, 17], there is no single fixed main memory area; rather, data dynamically move to the threads that access them. In such architectures, the notions of cache and main memory are blurred in a way that is hard to capture in CRF.

In this paper, we develop a unified representation of the memory models using the operational semantics methodology of *Abstract State Machines (ASM)* [2]. This makes it easier to compare the models. We also define a modified version of the *Location Consistency (LC)* memory model [7], in ASM terms, and prove that it is identical to the Manson-Pugh JMM. Finally, we prove that the CRF JMM is more restrictive than either the Manson-Pugh JMM or the LC JMM. Due to space constraints, we are unable to cover all Java memory model issues here. In particular, we do not consider *constructors* or *final fields*, which are given special treatment in both the Manson-Pugh and CRF JMMs. For details on these issues, see the M.S. Thesis by Awhad [3].

2 Overview of the New Java Memory Models

The new Java memory models [20, 21] preserve the fundamental ideas of the original [12]. Each *thread* executes Java bytecode instructions sequentially, with

writeNormal (Write$< v, w, g >$): overwritten$_t$ $\cup =$ previous$_t(v)$
previous$_t$ $+= < v, w, g >$
allWrites $+= < v, w, g >$

readNormal (Variable v): Choose $< v, w, g >$ from allWrites(v) - overwritten$_t(v)$
Return w

Fig. 1. Sample rules from the Manson-Pugh model [21]

read (load) and *write (store)* operations performed on a number of *variables*. A variable is a location at which *values* are stored. In a shared memory system, there may be multiple instantiations of a single variable (*e.g.*, cached by various threads), so there may be multiple values associated with a single variable at any time. The programmer may specify certain variables as *volatile*; such variables obey additional consistency constraints.

Threads may also perform basic synchronization operations. Each object has an associated monitor, on which threads perform *lock (enter)* and *unlock (exit)* operations. Lock and unlock instructions are guaranteed to be nested appropriately: a thread may enter a monitor m and then enter another monitor n without exiting m, but in this case it must exit n before exiting m.

2.1 Manson-Pugh JMM

Manson and Pugh describe their JMM in terms of a global system that maintains a history of write operations. The global system operates sequentially: in each step, it atomically executes one instruction from a thread. Execution involves firing an operational-style rule, similar to an ASM rule. Examples are shown in Figure 1.

Writes are flagged as *previous* or *overwritten* with respect to a given thread. Manson and Pugh informally describe these terms as follows: "[f]or each thread t, at any given step, overwritten$_t$ is the set of writes that thread t knows are overwritten and previous$_t$ is the set of all writes that thread t knows occurred previously" [21]. Aside from its anthropomorphic nature, this characterization is a little misleading: for instance, it seems unintuitive that t may read a value it did not "know" occurred previously, yet in the Manson-Pugh JMM it is possible for a thread t to read a value written by a write not in previous$_t$.

The significance of previous and overwritten is this: any write marked as overwritten by a thread t does not have a readable value according to t. A local write by t is immediately marked as previous according to t. A remote write is marked as previous according to t only after some synchronization between t and the writer. A write is marked as overwritten according to t only if it has already been marked as previous and another write intervenes (either a local write by t or a remote write followed by synchronization with t). Thus every write considered overwritten by t is also considered previous by t.

Writes are also marked as previous or overwritten with respect to each monitor. A lock operation has the effect of taking all writes that are previous or overwritten with respect to the locked monitor m and marking them similarly with respect to the locking thread t. An unlock has a similar effect, marking all previous or overwritten writes with respect to the unlocking thread t and marking them similarly with respect to the unlocked monitor m.

Volatile variables are different in that there is a single value marked readable for all threads and this globally readable value is the only readable value. The level of consistency is higher; all threads agree on a single value as the unique readable value. Furthermore, volatile variables are similar to threads and monitors in that they mark certain writes as previous or overwritten. Operations on volatile variables are similar to lock and unlock operations. A volatile read, like a lock, takes all writes marked previous or overwritten by the read variable and marks them previous or overwritten by the reading thread. A volatile write, like an unlock, takes all writes marked previous or overwritten by the writing thread and marks them previous or overwritten by the written variable.

2.2 Commit-Reconcile-Fence (CRF) JMM

Maessen, Arvind and Shen give a memory model for Java in terms of a translation schema from Java bytecode instructions to CRF instructions. The instruction set of the CRF model is much richer and finer-grained than that of Java bytecode. Consequently, each bytecode instruction corresponds to a sequence of CRF instructions. For instance, the CRF translation of a Java *load* instruction includes a *reconcile* instruction (ensuring a fresh value for the load), followed by a *loadl (load-local)* instruction (ensuring that a value is in the thread's cache). A Java store instruction has as its translation a *storel (store-local)* (ensuring that the new value is stored locally) and a following *commit* (ensuring the value is propagated to main memory). *Fence* instructions are also added in the translation to prevent certain kinds of instruction reordering.

Once this translation is done, CRF instruction can be reordered, modulo certain constraints. The abstract machine executes the resulting sequence of pending CRF instructions, each labeled with a unique result tag. When an instruction is executed, its result is associated with the result tag in a completion map.

Each thread has of a sequence of pending instructions, a set of completed instructions and a cache which maps addresses to values tagged with a state of either Clean or Dirty. The contents of the cache can be moved atomically to and from the main memory. Cache-to-cache data movement is possible only via the main memory. The model has two kinds of rules: *local rules* that execute CRF instructions and act purely on the cache; and *background rules* that move values to and from the main memory. Figure 2 shows some of these rules. Typically, local rules do little in terms of state updates; their primary task is to block until certain thread-local conditions are met. It is the duty of the background rules to effect the state changes that allow local operations to complete.

Local Rules:
$(r = Storel(a, v); I, comp, cache[a := -, -]) \Rightarrow (I, r = \surd/comp, cache[a := v, Dirty])$

$(r = Loadl(a); I, comp, cache[a := v, s]) \Rightarrow (I, r = v/comp, cache[a := v, s])$

$(r = Commit(a, v); I, comp, cache) \Rightarrow (I, r = \surd/comp, cache)$
where a is not in *cache* or a is $Clean$

$(r = Reconcile(a, v); I, comp, cache) \Rightarrow (I, r = \surd/comp, cache)$
where a is not in *cache* or a is $Dirty$

Background Rules:
$(I, comp, cache[a := v, s]) \Rightarrow (I, comp, cache)$
where s is $Clean$ (Eject a cache entry)

$(I, comp, cache)/threads, memory[a := v] \Rightarrow$
$(I, comp, cache[a := v, Clean])/threads, memory[a := v])$
where *cache* contains no mapping for a (Fetch a value from memory)

$(I, comp, cache[a := v, Dirty])/threads, memory[a := -] \Rightarrow$
$(I, comp, cache[a := v, Clean])/threads, memory[a := v]$ (Write back to memory)

Fig. 2. Sample rules from the CRF JMM [20]

Global states are defined as terms, and updates to global states are represented by rewriting terms, following the style of Shen and Arvind [24]. Examples are shown in Figure 2. This leads to rather unwieldy notation; even though each state change is typically quite local in scope, it requires an entire rewriting of the term. Thus for the nonexpert user, it can be difficult to find where the state is changing at each step. This is in contrast to ASM's locality principle for dynamics [5], which allows us to concentrate only on the changes to the states.

2.3 Location Consistency JMM

The Location Consistency (LC) memory model, proposed by Gao and Sarkar [7], was not proposed specifically as a memory model for Java, but in this paper we define a version of it appropriate for Java. In the model, the state of a memory location is described as a partially ordered set of instruction executions. Whenever an instruction is executed, a record of the operation is added to the partially ordered set of operations.

Chains of operations are formed in two ways: by a thread making a series of local operations, in which case the operation instances are linearly ordered, and by synchronization operations, where the operations of a locking thread are ordered with respect to operations performed by the previous unlocking thread. When a location is read, the value returned is the value associated with a maximal write in the partially ordered set for that location. Since the history

is a partial order rather than a total order, there may be multiple maximal write values that are legal candidates for a read operation.

LC has been formalized using ASM [28]. However, it is not exactly appropriate to Java, since LC lock operations are performed on locations (variables) rather than monitors, and a lock on a location affects the visibility of writes only on that location [27]. Thus we present a modified version of LC in which synchronization operations affect the visibility of writes to all variables.

3 ASM for the Manson-Pugh Model

In the ASM JMM_{MP}, representing the Manson-Pugh JMM, there are two types of agent. Thread agents (members of the universe Thread) execute Java instructions (members of the universe Instruction). These instructions are of four types: read, write, lock and unlock. A write operation generates a write instance (a member of the universe WriteEvent), which is added to the system history. A variable (a member of the universe Variable) and a value (a member of the universe Value) are associated with the write instance. A read operation generates a read instance (a member of the universe Read), selects a write instance that is legally readable, and reads the value of that write instance. A lock or unlock operation performs a synchronization with a given monitor (a member of the universe Monitor). Monitors are agents that control thread ownership.

As long as a Thread is not waiting for ownership of a monitor, it starts work on its next instruction, given by the monitored function currInst. The functions type, var, monitor, and val give the various parameters associated with each instruction. Read, write, and unlock operations are atomic, while a lock operation may require a complementary action by a Monitor agent, granting ownership to the requested monitor. Each write, lock, and unlock action updates the functions previous? and overwritten? appropriately. The Thread module is given in Fig. 3.

Monitors grant exclusive access to one thread at a time, and a thread may build up multiple locks on a single monitor, in which case it needs to perform multiple unlocks before any other thread can be granted a lock on the monitor. If some thread already has a claim on a monitor, then that thread may either increment or decrement its number of locks. If no thread has a claim, the monitor agent may choose a thread to grant a claim to. The Monitor module is given in Fig. 4. When a thread t is granted a lock on a monitor m, then the number of locks by t on monitor m (given by function locks) is incremented.

A write to a volatile variable v updates the globally visible value for v. Similarly, a read of a volatile variable simply uses this unique value; there is no choice involved. The previous? and overwritten? functions are updated similarly to the lock and unlock operations. The special rules for volatile variables are given in Fig. 5.

```
module Thread:
let inst = Self.currInst
    case inst.type of
    Read: Read inst.var
    Write: Write inst.val to inst.var
    Lock: Lock inst.monitor
    Unlock: Unlock inst.monitor

rule Read v:                                        (Perform a read of variable v)
if VolatileVariable(v) then Read volatile v
else
    extend Read with r
        choose w: WriteEvent: w.var = v and not w.overwritten?(Self)
            r.var := v
            r.val := w.val

rule Write val to v:                                (Write value val to variable v)
if VolatileVariable(v) then Write val to volatile v
else
    extend WriteEvent with w
        w.var := v
        w.val := val
        w.previous?(Self) := true
    do-forall w: WriteEvent: w.var = v and w.previous?(Self)
        w.overwritten?(Self) := true

rule Lock m:                                        (Get a lock on monitor m)
if Self.locks(m)≠undef then
    Self.locks(m) := Self.locks(m)+1
    do-forall w: WriteEvent: w.previous?(m)
        w.previous?(Self) := true
    do-forall w: WriteEvent: w.overwritten?(m)
        w.overwritten?(Self) := true

rule Unlock m:                                      (Release a lock on monitor m)
Self.locks(m) := Self.locks(m)-1
do-forall w: WriteEvent: w.previous?(Self)
    w.previous?(m) := true
do-forall w: WriteEvent: w.overwritten?(Self)
    w.overwritten?(m) := true
```

Fig. 3. JMM_{MP}: Thread module

4 ASM for the CRF Model

The ASM JMM_{CRF}, representing the CRF JMM, deals with instructions at two levels of abstraction: Java instructions, as in JMM_{MP}, and CRF instructions (members of the universe CRFInstruction). The CRF JMM uses "Load",

```
module Monitor:
if (∀t: Thread) t.locks(Self)=undef then
    choose t: Thread
        t.locks(Self) := 0
else
    choose t: Thread: t.locks(Self)=0
        t.locks(Self) := undef
```

Fig. 4. JMM_{MP}: Monitor module

```
rule Read volatile v:                          (Perform a read of volatile variable v)
extend Read with r
    r.var := v
    r.val := v.val
do-forall w: WriteEvent: w.previous?(v)
    w.previous?(Self) := true
do-forall w: WriteEvent: w.overwritten?(v)
    w.overwritten?(Self) := true

rule Write val to volatile v:                  (Write value val to volatile variable v)
extend WriteEvent with w
    w.var := v
    v.val := v
    do-forall w: WriteEvent: w.previous?(Self)
        w.previous?(v) := true
    do-forall w: WriteEvent: w.overwritten?(Self)
        w.overwritten?(v) := true
```

Fig. 5. JMM_{MP}: Rules for VolatileVariables

```
module Thread:
choose among
    Add Java instruction
    Reorder CRF instructions
    Execute CRF instruction
    Perform background operation
```

Fig. 6. JMM_{CRF}: Thread module

"Store", "Enter" and "Exit" as names for the Java instructions, as opposed to "Read", "Write", "Lock" and "Unlock" in the Pugh-Manson JMM. The actions of the ASM can be categorized as follows: add a Java instruction (translating it into appropriate CRF instructions), reorder CRF instructions, execute a CRF instruction, or initiate communication between a cache and main memory through a *background operation*. The Thread module for JMM_{CRF} is given in Fig. 6.

```
rule Proceed:      (Remove current CRF instruction from the pending instructions)
Self.CRFInsts := Self.CRFInsts.tail

rule Proceed when cache status not s:
(Proceed when the cache status of the current address is not s)
let v = Self.CRFInsts.head.addr
    if Self.cacheStatus(v) ≠ s then Proceed

rule Execute CRF instruction:
let in = Self.CRFInsts.head
    case in.type of
        Storel:     Self.cacheValue(in.addr) := in.val
                    Self.cacheStatus(in.addr) := Dirty
                    Proceed
        Loadl:      Proceed when cache status not Invalid
        Reconcile:  Proceed when cache status not Clean
        Commit:     Proceed when cache status not Dirty
        Fence:      Proceed
        Lock:       if Self.locks(in.monitor)≠undef then
                        Self.locks(in.monitor) := Self.locks(in.monitor)+1
                        Proceed
        Unlock:     Self.locks(in.monitor) := Self.locks(in.monitor)-1
                    Proceed
```

Fig. 7. JMM$_{CRF}$: Rule for execution of CRF instructions

CRF instructions are of the following types. A *storel (store local)* instruction writes a value to a thread's cache, without affecting main memory. The cache entry is flagged as *dirty*, indicating that its value must eventually be written back to main memory. Functions cacheValue and cacheStatus give the cache state for each thread and variable. The other CRF instructions do not affect the cache directly; rather, they block until certain cache conditions are met through execution of background operations. A *loadl (load local)* instruction completes only when an entry for the given variable has been loaded into main memory. A *reconcile* instruction completes only when the cache entry for the given variable is either nonexistent or dirty. This ensures that subsequent local loads use a relatively up-to-date value. A *commit* instruction completes only when a cache entry for the given variable is either nonexistent or clean. This ensures that any previous local store has had its value written back to main memory. Each CRF instruction has an associated type field. Each storel, loadl, reconcile, commit, lock, and unlock instruction has an addr field, giving the variable or monitor on which the instruction operates. In addition, each Storel instruction has a val field. Finally, a fence instruction has meaning only in terms of restricting certain kinds of instruction reordering; in terms of actual execution, a fence behaves as a no-op. Functions Storel, Loadl, *etc.* give CRF instructions

```
rule Perform background operation:
choose v: Variable
    case Self.cacheStatus(v) of
    Invalid:  Self.cacheValue(v) := v.memoryValue
              Self.cacheStatus(v) := Clean
    Clean:    Self.cacheValue(v) := undef
              Self.cacheStatus(v) := Invalid
    Dirty:    Self.cacheStatus(v) := Clean
              v.memoryValue := Self.cacheValue(v)
```

Fig. 8. JMM$_{CRF}$: Rule for background operations

```
rule Add CRF instructions in_1 ... in_k:
    Self.CRFInsts := Self.CRFInsts.addInst(in_1)
seq ...
seq Self.CRFInsts := Self.CRFInsts.addInst(in_k)

rule Add Java instruction:
let in = Self.currJavaInst
    case in.type of
    Load:   if VolatileVariable(in.var) then
                Add CRF instructions Fence(W,R,*V,in.var),Reconcile(in.var),
                    LoadI(in.var),Fence(R,R,in.var,*VR),Fence(W,R,in.var,*R)
            else Add CRF instructions Reconcile(in.var),LoadI(in.var)
    Store:  if VolatileVariable(in.var) then
                Add CRF instructions Fence(R,W,*VR,in.var),
                    Fence(W,W,*VR,in.var),StoreI(in.var,in.val),Commit(in.var)
            else Add CRF instructions StoreI(in.var,in.val),Commit(in.var)
    Enter:  Add CRF instructions Fence(W,W,*L,in.monitor),Lock(in.monitor),
                Fence(W,R,in.monitor,*VR),Fence(W,W,in.monitor,*VRL)
    Exit:   Add CRF instructions Fence(W,W,*VR,in.monitor),
                Fence(R,W,*VR,in.monitor),Unlock(in.monitor)
```

Fig. 9. JMM$_{CRF}$: Rules for adding CRF instructions

with the appropriate parameters. Fig. 7 shows the rule for executing a CRF instruction.

The interaction between cache and main memory that permits completion of these instructions is accomplished through so-called background operations. They do the following: in the case of a dirty cache entry, write the cached value to main memory and mark the entry clean; in the case of a clean cache entry, remove it; in the case of a nonexistent cache entry, create one and give it the current value stored in main memory. The function memoryValue gives the current value in main memory for each variable. Fig. 8 gives the rule for background operations.

As a Java instruction is added, it is translated into a sequence of CRF instructions (given by the function CRFInsts). The function addInst returns the CRFIn-

```
term a.inFenceAddr?(fa):                (Is address a included in fence address fa?)
a = fa
or (VolatileVariable(v) and fa ∈{*V,*VR,*VRL})
or (Variable(a) and not VolatileVariable(a) and fa ∈{*R,*VR,*VRL})
or (Monitor(a) and fa ∈{*L,*VRL})

term in1.fencedBefore?(in2):
(Is instruction in1 prevented from moving after (fence) instruction in2?)
in2.type=Fence and in1.type ∈{LoadI,Commit,Lock,Unlock}
and in1.addr.inFenceAddr?(in2.preAddr)
and (in1.type=LoadI ⇒ in2.preInst=R)
and (in1.type=Commit ⇒ in2.preInst=W)

term in2.fencedAfter?(in1):
(Is instruction in2 prevented from moving before (fence) instruction in2?)
in1.type=Fence and in2.type ∈{Reconcile,StoreI,Lock,Unlock}
and in2.addr.inFenceAddr?(in1.postAddr))
and (in2.type=Reconcile ⇒ in1.postOp=R)
and (in2.type=StoreI ⇒ in1.postOp=W)

term swappable?(in1, in2):
(Can instruction in1 be swapped with the following instruction in2?)
not in1.fencedBefore?(in2) and not in2.fencedAfter?(in1)
and in1.type=LoadI ⇒ not(in2.type=StoreI and in2.addr=in1.addr)
and in1.type=StoreI ⇒ not(in2.type ∈{LoadI,StoreI,Commit} and in2.addr=in1.addr)
and in1.type=Lock ⇒ not(in2.type=Unlock and in2.addr=in1.addr)
and in1.type=Reconcile ⇒ not(in2.type=LoadI and in2.addr=in1.addr)

rule Reorder CRF instructions:
choose n: Nat: Self.CRFInsts.at(n+1)≠undef
               and swappable?(Self.CRFInsts.at(n), Self.CRFInsts.at(n+1))
    Self.CRFInsts := Self.CRFInsts.swapAt(n)
```

Fig. 10. JMM$_{CRF}$: Rules for reordering CRF instructions

struction sequence achieved by appending a given CRFInstruction to a given sequence. This sequence can be permuted, with certain exceptions. For instance, a local load instruction on a given variable cannot be executed before a reconcile instruction on the same variable that precedes it in original program order; this would subvert the intent of the reconcile, to ensure that a fresh value is loaded in at the local load. The function swapAt takes a sequence of CRFInstructions and an index n into the sequence and returns the result of swapping the instructions at indices n and $n + 1$. Fig. 9 and Fig. 10 give the rules for adding Java instructions and reordering CRF instructions, respectively.

Fence instructions control reordering of other CRF instructions. Each fence has a *pre-instruction type* and a *post-instruction type*. A pre-instruction type of R indicates that the fence prevents certain kinds of read instruction from moving

after the fence, while a pre-instruction type of W means that certain writes cannot be moved after the fence. Similarly, the post-instruction type indicates what kinds of instruction (read or write) are prevented from moving before the fence. In addition, a fence has a *preaddress* and a *postaddress*. These restrict the scope of the fence to instructions on certain addresses. A fence (pre/post)address may be a single variable or monitor, or it may be a wildcard reference such as *V, *VR, or *VRL. The fence address *V refers to "all volatile variables", *VR to "all (regular and volatile) variables", and *VRL to "all variables and locks". Thus for instance, a fence with pre-instruction type W, post-instruction type R, preaddress *V, and postaddress v (for some variable v) disallows movement of preceding write instructions on volatile variables after the fence and disallows movement of following read instructions on variable v before the fence. The FenceAddress universe is defined as Variable $\cup$ Monitor $\cup$ {*V,*R,*L,*VR,*VRL}. Each Fence instruction has associated preInst, postInst, preAddr, and postAddr fields.

5 ASM for the Location Consistency JMM

We restrict the Location Consistency memory to make it appropriate to Java. The resulting ASM is called JMM$_{LC}$. The ASM is a modification of JMM$_{MP}$, involving modifications to the rules for reads, writes, locks and unlocks. The modified rules are shown in Fig. 11. Like the LC model, in JMM_{LC} the memory system is represented as an order $\prec$ on operation instances or *events* (members of the universe Event). As operations complete, events are generated and added to the order. The events issued by a single thread are ordered linearly. With a write to a given variable, a new write event is issued and ordered after the writing thread's local events on that variable. The function latest maintains the most recently issued events by each thread on each variable.

Events ordered by different threads may be ordered through lock and unlock actions. Each lock or unlock operation is seen as acting on all variables, so a lock or unlock event is ordered with respect to all events issued by the thread. This is where the LC JMM differs from the general LC model: an acquire or release operation in LC is performed on a particular location, and the resulting event is ordered only with regard to local events on that location. Similarly to the general model, each lock event on a given monitor is ordered after the previous unlock event on that monitor. The function latestUnlock maintains the most recently issued unlock events for each monitor.

Like the Manson-Pugh JMM, reading involves a choice of a legal write event. Here the order $\prec$ determines the legality of each write event. We use $\prec^+$ and $\prec^*$ to represent the transitive closure of $\prec$ and the reflexive, transitive closure of $\prec$, respectively. As long as there is no chain of the form $w \prec^+ w' \prec^* e_t$, where w and w' are write events and e_t is an event performed by thread t, w is a readable value according to t. Read events have no effect on the readability of subsequent write events, so they are not added to the order.

```
rule Order e after d:          (Make event e a successor of event d, in the relation ≺)
if d ≠ undef then d ≺ e := true

rule Read v:
extend ReadEvent with r
    choose w: Write:
            w.var = v and not (∃w′: WriteEvent) w ≺+ w′ ≺* Self.latest(w.var)
        r.var := v
        r.val := w.val

rule Write val to v:
extend WriteEvent with w
    w.var := v
    w.val := val
    Order w after Self.latest(v)
    Self.latest(v) := w

rule Lock m:
if Self.locks(m)≠undef then
    Self.locks(m) := Self.locks(m)+1
    extend LockEvent with ℓ
        Order ℓ after m.latestUnlock
        do-forall v: Variable
            Order ℓ after Self.latest(v)
            Self.latest(v) := ℓ

rule Unlock m:
extend UnlockEvent with u
    Self.locks(m) := Self.locks(m)-1
    do-forall v: Variable
        Order u after t.latest(v)
        Self.latest(v) := ℓ
    m.latestUnlock := u
```

Fig. 11. JMM_{LC}: Rules for thread operations

6 Comparisons of the Java Memory Models

We now use our ASMs to reason about the behavior of the JMMs. For brevity, we restrict our attention to non-volatile variables. We first show that the Manson-Pugh and LC JMMs are identical. Next, we show that the CRF JMM is more restrictive than either the Manson-Pugh or the LC JMM. We omit the proof details; the reader may find them in Awhad's M.S. thesis [3].

6.1 Manson-Pugh *vs.* LC

In both the Manson-Pugh and LC JMMs, the synchronization operations (lock and unlock) ensure that other threads are informed of previous write operations. However, threads may share values without synchronization; note that in the absence of all synchronization, all writes are visible to a thread (except any writes of its own that it has subsequently overwritten). Indeed, synchronization operations restrict the visibility of certain writes by certain threads. When writes are interleaved with synchronization operations, the resulting chains of write and synchronization operations may cause further writes to be considered overwritten. We introduce the notion of *synchronization chain* to capture this idea.

A synchronization chain begins with a write w and ends with an action by a thread or monitor x. (We express this as $w \rightsquigarrow x$.) The thread or monitor x is informed of the existence of w through a subsequent chain of synchronization operations, including an unlock action by the writer of w and a subsequent unlock operation on x (if x is a monitor) or lock operation by x (if x is a thread). (A trivial synchronization chain exists between w and the thread that performed w.) If another write w' forms part of the chain containing w (which we express as $w \rightsquigarrow w'$), w is considered overwritten by w' and is not a readable value.

Definition. Let w be a write event issued at a move W by a thread s. Let Y be any move after W.

- For thread t, $w \rightsquigarrow t$ at Y if (1) $s = t$, or
  (2) for some Monitor m, $w \rightsquigarrow m$ at a move X in (W,Y), and t locks m at X.
- For monitor m, $w \rightsquigarrow m$ at Y if (1) s unlocks m at a move X in (W,Y), or
  (2) for some Thread t, $w \rightsquigarrow t$ at a move X in (W,Y), and t unlocks m at X.
- For any WriteEvent w' issued by a Thread t at a move X, $w \rightsquigarrow w'$ if $w \rightsquigarrow t$ at X.

Lemma 1 *In a run of JMM_{MP}, let w be a write event issued at a move W, and let Y be any move after W. For any thread or monitor x, w.*previous?*(x) at Y if and only if $w \rightsquigarrow x$ at Y.*

Lemma 2 *In a run of JMM_{MP}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W. For any* Thread *or* Monitor *x, w.*overwritten?*(x) at Y if and only if there is a* WriteEvent *w' such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow x$ at Y.*

Next, we show that the notions of "previous" and "overwritten" have counterparts in the LC JMM, as shown in Fig. 12. A write is considered previous by a thread or monitor if it precedes the latest event by the thread/monitor, according to $\prec$. A write is considered overwritten if there is a chain including the write and a local event by the thread/monitor that includes another, intervening write.

Lemma 3 *In a run of JMM_{LC}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W. For any* Thread *or* Monitor *x, w.*LC-previous?*(x) at Y if and only if $w \rightsquigarrow x$ at Y.*

term w.LC-previous?(x):
Thread(x) $\Rightarrow$ $w \prec^* x$.latest(w.var)
and Monitor(x) $\Rightarrow$ $w \prec^+ x$.latestUnlock

term w.LC-overwritten?(x):
Thread(x) $\Rightarrow$ ($\exists w'$: WriteEvent) $w \prec^+ w' \prec^* x$.latest(w.var)
and Monitor(x) $\Rightarrow$ ($\exists w'$: WriteEvent) $w \prec^+ w' \prec^+ x$.latestUnlock

Fig. 12. Counterparts to previous? and overwritten? in the LC JMM

Lemma 4 *In a run of JMM_{LC}, let w be a* WriteEvent *issued at a move W, and let Y be any move after W.*

- *For any* Thread t, w.LC-overwritten?(t) *at Y if and only if there is a* WriteEvent w' *such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow t$ at Y.*
- *For any* Monitor m, w.LC-overwritten?(m) *at Y if and only if there is a* WriteEvent w' *such that $w \rightsquigarrow w'$ and $w' \rightsquigarrow m$ at Y.*

Finally, note that any write is readable in JMM_{LC} as long as LC-overwritten? evaluates to false, and any write is readable in JMM_{MP} as long as overwritten? evaluates to false. Let runs of JMM_{LC} and JMM_{MP} be equivalent if they involve the same operations by the same threads/monitors in the same order. We conclude the following:

Corollary *A value is readable in JMM_{LC} if and only if it is readable in an equivalent run of JMM_{MP}.*

6.2 Manson-Pugh/LC *vs.* CRF

Having established the equivalence of the Manson-Pugh JMM and the LC JMM, we now compare these two models to the CRF JMM. Here we face an additional level of complexity. In all three models, each thread services a sequence of Java instructions. But in the CRF JMM, the actions of reading and writing are done at the level of CRF instructions, which are derived from Java instructions and then possibly reordered. In reasoning about the values actually written and read (to local caches) in the CRF JMM, we must look at the level of CRF instructions. But to provide a comparison to the other models, we must relate the relevant CRF instructions back to the Java instructions that triggered them, bearing in mind that instruction reordering may occur between the issuing of the Java instruction and the execution of the CRF instruction.

To this end, we introduce some notation. Let m be a move at which a CRF instruction is executed. This instruction was generated at an earlier move, in which a Java instruction was issued and translated into CRF. We refer to the earlier move as $\overline{m}$. We then establish that the CRF JMM is at least as constrained as the other two models.

Lemma 5 *Let ll be a local load in a run of JMM_{CRF}, and let sl be the local store that wrote the value read at ll. Then there is no store operation S such that $\overline{sl} \rightsquigarrow S \rightsquigarrow t$ at $\overline{ll}$.*

Corollary *A value is readable in JMM_{CRF} only if it is readable in an equivalent run of JMM_{MP}.*

The converse of the above corollary is not true, as can be shown by a simple example. Let T_1 and T_2 be Threads accessing a common Variable x. The Threads execute the following instruction sequences:

T_1**:** write$(x, 1)$; read(x); read(x)
T_2**:** write$(x, 2)$

Assume that x=0 initially. It is easy to show that in any run of JMM_{CRF} with the above sequences of Java instructions, if T_1 first reads 2, it cannot then read 1. First, note that T_1's Storel and Loadl instructions follow the program order of the original Java instructions, since the swappable? predicate prevents the Storel from being ordered after the initial Loadl.

At T_1's first loadl, T_1.cacheValue(x)=2. This is only possible if a fetch (background) operation earlier updated T_1.cacheValue(x) := x.memoryValue. At the time of this fetch, T_1.cacheStatus(x)=Invalid. Thus an earlier writeback (background) operation must have updated T_1.cacheStatus(x) := Clean, followed by an ejection updating T_1.cacheStatus(x) := Invalid. After this writeback, T_1.cacheStatus(x)=Clean, so T_1 never performs another writeback. Since x.memoryValue=2 at the time of T_1's fetch, a writeback operation must have updated x.memoryValue := T_2.cacheValue(x), and this writeback must have followed T_1's writeback. Thus x.memoryValue remains 2, and the value 1 is no longer available to read.

No such constraint exists for the Manson-Pugh JMM. Let w_1 and w_2 be the WriteEvents issued by T_1 and T_2, respectively. Since there is no synchronization between T_1 and T_2, $w_2 \not\rightsquigarrow T_1$ at the time of T_1's second read, so there is no w' such that $w_1 \rightsquigarrow w' \rightsquigarrow T_1$. By Lemma 2, w_1 is readable.

7 Conclusion

In this paper, we have presented formal semantics of the two proposed replacements for the Java memory model, as well as a variant of Location Consistency appropriate to Java, using the common formalism of Abstract State Machines. Using a single specification technique allows us to compare the models easily. Since the original descriptions of the models are so different, in devising a unified formalization we run the risk of what Börger calls the "formal system straitjacket" [5]: by forcing the specifications to conform to a rigid format, we may lose the essence of the original descriptions. However, ASM's descriptive flexibility allows us to keep our specifications close to the originals.

Researchers involved with the Java memory model have recognized the importance of precise specification. Several formalizations of the original Java memory model have appeared [10, 14, 23]. Yang, Gopalakrishnan, and Lindstrom [30, **?**] give specifications of the new Java memory models in terms of their Uniform Memory Model (UMM) framework, based on an operational approach similar to ASM. The authors demonstrate the benefits of formal specification by discovering several mistakes in the original description of the Manson-Pugh proposal. This work is designed for automated verification, and as a result the translation from the original descriptions to UMM is rather involved. For instance, the Manson-Pugh Java memory model is expressed in terms closer to our version of Location Consistency. Integrating our high-level specifications with a lower-level framework like UMM seems an interesting direction for future research.

Complicating matters is the continuing flux of the Java memory model debate. Manson and Pugh are drafting a new memory model semantics, and there are at least two additional alternatives in preparation. Although an official decision process is in place, there is no official successor to the original memory model.

Predicting the behavior of multithreaded Java applications on various platforms is difficult; even experts can fail to catch subtle but important errors [9, 16]. For Java programmers using multithreading, the ability to simulate executions on different architectures would be a great advantage. Awhad's M.S. thesis work will provide executable versions of our ASM specifications, using the XASM tool [29]. Of course, this work ignores the details of Java statements and expressions, as well as important issues like class loading and object initialization. There exist executable ASM specifications of Java that include such details [26, 18], but they do not deal with the underlying memory model. Our next goal is to provide a complete specification of Java that integrates the memory model with other features of the language.

Acknowledgments

We thank Guy Tremblay, for his comments on an earlier draft of this paper, and José Nelson Amaral, for his thoughts on Location Consistency.

References

[1] Adve, S. V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. Research Report 95/7, Digital Western Research Laboratory (1995) 166, 167
[2] ASM Home Page. `http://www.eecs.umich.edu/gasm/` 168
[3] Awhad, V.: A Unified Formal Specification and Analysis of the New Java Memory Models. M. S. Thesis, Michigan Technological Univ. (2002) 168, 179
[4] Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E., Randall, K. H.: An Analysis of DAG-consistent Distributed Shared-Memory Algorithms. Proc. ACM SPAA, (1996) 297–308 167

[5] Börger, E.: Why Use Evolving Algebras for Hardware and Software Engineering? In: Bartosek, M., Staudek, J., Wiedermann, J. (eds.): SOFSEM '95: 22nd Seminar on Current Trends in Theory and Practice of Informatics. LNCS 1012, Springer-Verlag (1995) 236–271 171, 182
[6] Cenciarelli, P., Knapp, A., Reus, B., Wirsing, M.: ¿From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In: Johnson, M. (ed.): Algebraic Methodology and Software Technology. Springer-Verlag (1997) 75–90
[7] Gao, G. R., Sarkar, V.: Location Consistency — A New Memory Model and Cache Consistency Protocol. IEEE Trans. on Comp. 49(8) (2000) 798–813 167, 168, 171
[8] Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., Hennessy, J.: Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. Proc. ISCA (1990) 15–26 167
[9] Goetz, B.: Double-Checked Locking: Clever, but Broken. JavaWorld 6(2) (2001) 183
[10] Gontmakher, A., Schuster, A.: Java consistency: Non-operational characterizations for Java memory behavior. ACM Trans. on Comp. Sys. 18(4) (2000) 333–386 183
[11] Goodman, J. R.: Cache Consistency and Sequential Consistency. Technical Report 1006, Computer Science Dept., U. of Wisconsin–Madison (1989) 167
[12] Gosling, J., Joy, B., Steele, G.: The Java Language Specification. Addison-Wesley (1996) 168
[13] Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, N., Saphir, W., Snir, M.: MPI: The Complete Reference. MIT Press (1998) 167
[14] Gurevich, Y., Schulte, W., Wallace, C.: Investigating Java concurrency Using Abstract State Machines. In: Gurevich, Y., Kutter, P. W., Odersky, M., Thiele, L. (eds.): Abstract State Machines: Theory and Applications. LNCS 1912. Springer-Verlag (2000) 151–176 168, 183
[15] Hagersten, E., Landin, A., Haridi, S.: DDM — A Cache Only Memory Architecture. IEEE Computer 25(9) (1992) 44–54 168
[16] Holub, A.: Warning! Threading in a Multiprocessor World. JavaWorld 6(2) (2001) 183
[17] Joe, T.: COMA-F: A Non-Hierarchical Cache Only Memory Architecture. Ph.D. Thesis, Stanford Univ. (1995) 168
[18] Kutter, P. W.: Montages — Engineering of Programming Languages. Ph.D. Thesis, Eidgenössische Technische Hochschule Zürich (2002) 183
[19] Lamport, L.: How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. IEEE Transactions on Computers C-28(9) (1979) 690–691 167
[20] Maessen, J.-W., Arvind, Shen, X.: Improving the Java Memory Model Using CRF. Proc. OOPSLA (2000) 1–12 168, 171
[21] Manson, J., Pugh, W.: Multithreaded Semantics for Java. CS Technical Report 4215, Univ. of Maryland (2001) 168, 169
[22] Pugh, W.: Fixing the Java Memory Model. Proc. ACM Java Grande (1999) 168
[23] Roychoudhury, A., Mitra, T.: Specifying Multithreaded Java Semantics for Program Verification. Proc. ICSE (2002) 183
[24] Shen, X., Arvind: Specification of Memory Models and Design of Provably Correct Cache Coherent Protocols. CSG Memo 398, Laboratory for Computer Science, MIT (1997) 171

[25] Shen, X., Arvind, Rudolph, L.: Commit-Reconcile & Fences (CRF): A New Memory Model for Architects and Compiler Writers. Proc. ISCA (1999) 150–161 167, 168

[26] Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine: Definition, Verification, Validation. Springer-Verlag (2001) 183

[27] Wallace, C., Tremblay, G., Amaral, J. N.: On the Tamability of the Location Consistency Memory Model. Proc. PDPTA (2002) 172

[28] Wallace, C., Tremblay, G., Amaral, J. N.: An Abstract State Machine Specification and Verification of the Location Consistency Memory Model and Cache Protocol. J. Universal Computer Science 7(11) (2001) 1088–1112 172

[29] XASM Home Page. `http://www.xasm.org/'` 183
`\bibitem{YanGopLin:02}`Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Formalizing the Java Memory Model for Multithreaded Program Correctness and Optimization. Technical Report UUCS-02-011, Univ. of Utah.

[30] Yang, Y., Gopalakrishnan, G., Lindstrom, G.: Analyzing the CRF Java Memory Model. Proc. Asia-Pacific Software Engineering Conference (2001) 21-28 183

Modelling Conditional Knowledge Discovery and Belief Revision by Abstract State Machines*

Christoph Beierle and Gabriele Kern-Isberner

Praktische Informatik VIII - Wissensbasierte Systeme
Fachbereich Informatik, FernUniversität Hagen
58084 Hagen, Germany
{christoph.beierle,gabriele.kern-isberner}@fernuni-hagen.de

Abstract. We develop a high-level ASM specification for the CONDOR system that provides powerful methods and tools for managing knowledge represented by conditionals. Thereby, we are able to elaborate crucial interdependencies between different aspects of knowledge representation, knowledge discovery, and belief revision. Moreover, this specification provides the basis for a stepwise refinement development process of the CONDOR system based on the ASM methodology.

1 Introduction

Commonsense and expert knowledge is most generally expressed by rules, connecting a precondition and a conclusion by an *if-then*-construction. If-then-rules are more formally denoted as conditionals, and often they occur in the form of probabilistic (quantitative) conditionals like "*Students are young with a probability of (about) 80 %*" and "*Singles (i.e. unmarried people) are young with a probability of (about) 70 %*", where this commonsense knowledge can be expressed formally by $\{(young|student)[0.8], (young|single)[0.7]\}$. In another setting, qualitative conditionals like $(expensive|Mercedes)[n]$ are considered where $n \in \mathbb{N}$ indicates a degree of plausibility for the conditional "*Given that the car is a Mercedes, it is expensive*".

The crucial point with conditionals is that they carry generic knowledge which can be applied to different situations. This makes them most interesting objects in Artificial Intelligence, in theoretical as well as in practical respect. Within the CONDOR project (Conditionals - discovery and revision), we develop methods and tools for discovery and revision of knowledge expressed by conditionals. Our aim is to design, specify, and develop the ambitious CONDOR system using Abstract State Machines, based on previous experiences with the ASM approach and using tools provided by the ASM community.

Figure 1 provides a bird's-eye view of the CONDOR system. CONDOR can be seen as an agent being able to take rules, evidence, queries, etc., from the environment and giving back sentences he believes to be true with a degree of

* The research reported here was partially supported by the DFG – Deutsche Forschungsgemeinschaft (grant BE 1700/5-1).

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 186–203, 2003.
© Springer-Verlag Berlin Heidelberg 2003

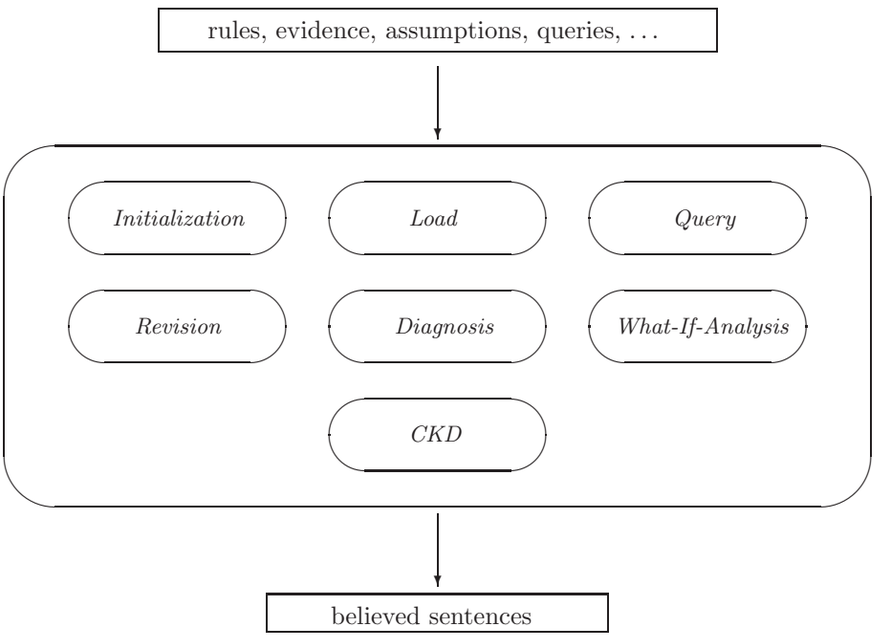


Fig. 1. A bird's-eye view of the CONDOR-Systems

certainty. Basically, these degrees of belief are calculated from the agent's current epistemic state which is a representation of his cognitive state at the given time. The agent is supposed to live in a dynamic environment, so he has to adapt his epistemic state constantly to changes in the surrounding world and to react adequately to new demands.

In this paper, we develop a basic ASM, denoted by CONDORASM, providing the top-level functionalities of the CONDOR system as they are indicated by the buttons in Figure 1. Using this ASM, we are not only able to decribe precisely the common functionalities for dealing with both quantitative and qualitative approaches. We also work out crucial interdependencies between e.g. inductive knowledge representation, knowledge discovery, and belief revision in a conditional setting. Moreover, CONDORASM provides the basis for a stepwise refinement development process of the CONDOR system.

The rest of this paper is organized as follows: In Section 2, we provide a very brief introduction to qualitative and quantitative logics. In Section 3, the universes of CONDORASM and its overall structure are introduced, while in Section 4 its top-level functions are specified. Section 5 contains some conclusions and points out further work. In Appendix A, we summarize the universes, functions, constraints, and transition rules for CONDORASM that are developed throughout this paper.

2 Background: Qualitative and Quantitative Logic in a Nutshell

We start with a propositional language $\mathcal{L}$, generated by a finite set Σ of atoms $a, b, c, \ldots$. The formulas of $\mathcal{L}$ will be denoted by uppercase roman letters $A, B, C, \ldots$. For conciseness of notation, we will omit the logical *and*-connector, writing AB instead of $A \wedge B$, and barring formulas will indicate negation, i.e. $\overline{A}$ means $\neg A$. Let Ω denote the set of possible worlds over $\mathcal{L}$; Ω will be taken here simply as the set of all propositional interpretations over $\mathcal{L}$ and can be identified with the set of all complete conjunctions over Σ. $\omega \models A$ means that the propositional formula $A \in \mathcal{L}$ holds in the possible world $\omega \in \Omega$.

By introducing a new binary operator $|$, we obtain the set $(\mathcal{L} \mid \mathcal{L}) = \{(B|A) \mid A, B \in \mathcal{L}\}$ of conditionals over $\mathcal{L}$. $(B|A)$ formalizes "*if* A *then* B" and establishes a plausible, probable, possible etc connection between the *antecedent* A and the *consequent* B. Here, conditionals are supposed not to be nested, that is, antecedent and consequent of a conditional will be propositional formulas.

To give appropriate semantics to conditionals, they are usually considered within richer structures such as *epistemic states*. Besides certain (logical) knowledge, epistemic states also allow the representation of preferences, beliefs, assumptions etc of an intelligent agent. Basically, an epistemic state allows one to compare formulas or worlds with respect to plausibility, possibility, necessity, probability etc.

Well-known qualitative, ordinal approaches to represent epistemic states are Spohn's *ordinal conditional functions, OCFs*, (also called *ranking functions*) [Spo88], and *possibility distributions* [BDP92], assigning degrees of plausibility, or of possibility, respectively, to formulas and possible worlds. In such qualitative frameworks, a conditional $(B|A)$ is valid (or *accepted*), if its confirmation, AB, is more plausible, possible etc. than its refutation, $A\overline{B}$; a suitable degree of acceptance is calculated from the degrees associated with AB and $A\overline{B}$.

In a quantitative framework, most appreciated representations of epistemic states are provided by *probability functions* (or *probability distributions*) $P : \Omega \to [0, 1]$ with $\sum_{\omega \in \Omega} P(\omega) = 1$. The probability of a formula $A \in \mathcal{L}$ is given by $P(A) = \sum_{\omega \models A} P(\omega)$, and the probability of a conditional $(B|A) \in (\mathcal{L} \mid \mathcal{L})$ with $P(A) > 0$ is defined as $P(B|A) = \dfrac{P(AB)}{P(A)}$, the corresponding conditional probability. Note that, since $\mathcal{L}$ is finitely generated, Ω is finite, too, and we only need additivity instead of σ-additivity.

3 The Formal Framework of CondorASM

3.1 Universes

On a first and still very abstract level we do not distinguish between qualitative and quantitative conditionals. Therefore, we use Q as the universe of qualitative and quantitative scales.

The universe Σ of propositional variables provides a vocabulary for denoting simple facts. The universe Ω contains all possible worlds that can be distinguished using Σ. $Fact_{\mathcal{U}}$ is the set of all (unquantified) propositional sentences over Σ, i.e. $Fact_{\mathcal{U}}$ consists of all formulas from $\mathcal{L}$. The set of all (unquantified) conditional sentences from $(\mathcal{L} \mid \mathcal{L})$ is denoted by $Rule_{\mathcal{U}}$.

The universe of all sentences without any qualitative or quantitative measure is given by

$$Sen_{\mathcal{U}} = Fact_{\mathcal{U}} \cup Rule_{\mathcal{U}}$$

with elements written as A and $(B|A)$, respectively. Additionally, $SimpleFact_{\mathcal{U}}$ denotes the set of simple facts $\Sigma \subseteq Fact_{\mathcal{U}}$, i.e. $SimpleFact_{\mathcal{U}} = \Sigma$.

Analogously, in order to take quantifications of belief into account, we introduce the universe $Sen_{\mathcal{Q}}$ of all qualitative or quantitative sentences by setting

$$Sen_{\mathcal{Q}} = Fact_{\mathcal{Q}} \cup Rule_{\mathcal{Q}}$$

whose elements are written as $A[x]$ and $(B|A)[x]$, respectively, where $A, B \in Fact_{\mathcal{U}}$ and $x \in Q$. For instance, the measured conditional $(B|A)[x]$ has the reading *if A then B with degree of belief x*. The set of measured simple facts is denoted by $SimpleFact_{\mathcal{Q}} \subseteq Fact_{\mathcal{Q}}$.

The universe of epistemic states is given by $EpState \subseteq \{\Psi \mid \Psi : \Omega \to Q\}$. We assume that each $\Psi \in EpState$ uniquely determines a function (also denoted by Ψ) $\Psi : Sen_{\mathcal{U}} \to Q$. For instance, in a probabilistic setting, for $\Psi = P : \Omega \to [0,1]$ we have $P(A) = \sum_{\omega \models A} P(\omega)$ for any unquantified sentence $A \in Sen_{\mathcal{U}}$.

Finally, there is a binary satisfaction relation (modelled by its characteristic function in the ASM framework) $\models_{\mathcal{Q}} \subseteq EpState \times Sen_{\mathcal{Q}}$ such that $\Psi \models_{\mathcal{Q}} S$ means that the state Ψ satisfies the sentence S. Typically, Ψ will satisfy a sentence like $A[x]$ if Ψ assigns to A the degree x ("*in Ψ, A has degree of probability / plausibility x*").

In this paper, our standard examples for epistemic states are probability distributions, but note that the complete approach carries over directly to the ordinal framework (see eg. [KI01a]).

Example 1. In a probabilistic setting, conditionals are interpreted via conditional probability. So for a probability distribution P, we have $P \models_{\mathcal{Q}} (B|A)[x]$ iff $P(B|A) = x$ (for $x \in [0,1]$).

3.2 Overall Structure

In the CondorASM, the agent's current epistemic state is denoted by the controlled nullary function[1]

$$currstate : EpState$$

The agents beliefs returned to the environment can be observed via the controlled function

$$believed_sentences : \mathcal{P}(Sen_{\mathcal{Q}})$$

[1] For a general introdcution to ASMs and also to stepwise refinement using ASMs see e.g. [Gur95] and [SSB01]; in particular, we will use the classification of ASM functions – e.g. into controlled or monitored functions – as given in [SSB01].

input type		monitored nullary function
$\mathcal{P}(Sen_{\mathcal{Q}})$	:	*rule_base* *new_information* *assumptions*
$\mathcal{P}(Sen_{\mathcal{U}})$	:	*queries* *goals*
$\mathcal{P}(Fact_{\mathcal{Q}})$	:	*evidence*
$\mathcal{P}(SimpleFact_{\mathcal{U}})$	:	*diagnoses*
EpState	:	*stored_state* *distribution*
RevisionOp	:	*rev_op*

Fig. 2. Monitored function in CondorASM

with $\mathcal{P}(S)$ denoting the power set of S.

As indicated in Figure 1, there are seven top-level functions that can be invoked, ranging from initialization of the system to the automatic discovery of conditional knowledge (CKD). Thus, we have a universe

$$\begin{aligned} \mathit{WhatToDo} \;=\; \{\, &\mathit{Initialization}, \mathit{Load}, \mathit{Query}, \mathit{Revision}, \\ &\mathit{Diagnosis}, \mathit{What\text{-}If\text{-}Analysis}, \mathit{CKD} \,\} \end{aligned}$$

The nullary interaction function

$$do : \mathit{WhatToDo}$$

is set by the environment in order to invoke a particular function and is reset by CondorASM on executing it.

The appropriate inputs to the top-level functions are modelled by monitored nullary functions set by the environment. For instance, simply querying the system takes a set of (unquantified) sentences from $Sen_{\mathcal{U}}$, asking for the degree of belief for them. Similarly, the *What-If-Analysis* realizes hypothetical reasoning, taking a set of (quantified) sentences from $Sen_{\mathcal{Q}}$ as assumptions, together with a set of (unquantified) sentences from $Sen_{\mathcal{U}}$ as goals, asking for the degree of belief for these goals under the given assumptions. Figure 2 summarizes all monitored functions serving as inputs to the system; their specific usage will be explained in detail in the following section along with the corresponding top-level functionalities.

4 Top-Level Functions in the Condor-System

4.1 Initialization

In the beginning, a prior epistemic state has to be built up on the basis of which the agent can start his computations. If no knowledge at all is at hand, simply

the uniform epistemic state, modelled by the nullary function

$$\mathit{uniform} : \mathit{EpState}$$

is taken to initialize the system. For instance, in a probabilistic setting, this corresponds to the uniform distribution where everything holds with probability 0.5.

If, however, default knowledge or a set of probabilistic rules is at hand to describe the problem area under consideration, an epistemic state has to be found to appropriately represent this prior knowledge. To this end, we assume an inductive representation method to establish the desired connection between sets of sentences and epistemic states. Whereas generally, a set of sentences allows a (possibly large) set of models (or epistemic states), in an inductive formalism we have a function

$$\mathit{inductive} : \mathcal{P}(\mathit{Sen}_{\mathcal{Q}}) \rightarrow \mathit{EpState}$$

such that $\mathit{inductive}(S)$ selects a unique, "best" epistemic state from all those states satisfying S. Starting with the no-knowledge representing state *uniform* can be modelled by providing the empty set of rules since the constraint

$$\mathit{uniform} = \mathit{inductive}(\emptyset)$$

must hold.

Thus, we can initialize the system with an epistemic state by providing a set of (quantified) sentences S and generating a full epistemic state from it by inductively completing the knowledge given by S. For reading in such a set S, the monitored nullary function $\mathit{rule_base} : \mathcal{P}(\mathit{Sen}_{\mathcal{Q}})$ is used:

if $\mathit{do} = \mathit{Initialization}$
then $\mathit{currstate} := \mathit{inductive}(\mathit{rule_base})$
$\mathit{do} := \mathit{undef}$

Selecting a "best" epistemic state from all all those states satisfying a set of sentences S is an instance of a general problem which we call the *representation problem* (cf. [BKI02a]). There are several well-known methods to model such an inductive formalism, a prominent one being the *maximum entropy* approach.

Example 2. In a probabilistic framework, the *principle of maximum entropy* associates to a set $\mathcal{S}$ of probabilistic conditionals the unique distribution $P^* = \mathit{MaxEnt}(\mathcal{S})$ that satisfies all conditionals in $\mathcal{S}$ and has maximal entropy, i.e., $\mathit{MaxEnt}(\mathcal{S})$ is the (unique) solution to the maximization problem

$$\max H(P') = -\sum_{\omega} P'(\omega) \log P'(\omega) \tag{1}$$
$$\text{s.t. } P' \text{ is a probability distribution with } P' \models \mathcal{S}.$$

The rationale behind this is that $\mathit{MaxEnt}(\mathcal{S})$ represents the knowledge given by $\mathcal{S}$ most faithfully, i.e. without adding information unnecessarily (cf. [Par94, PV97, KI98]).

We will illustrate the maximum entropy method by a small example.

Example 3. Consider the three propositional variables a - *being a student*, b - *being young*, and c - *being unmarried*. *Students* and *unmarried people* are mostly *young*. This commonsense knowledge an agent may have can be expressed probabilistically e.g. by the set $\mathcal{S} = \{(b|a)[0.8], (b|c)[0.7]\}$ of conditionals. The *MaxEnt*-representation $P^* = MaxEnt(\mathcal{S})$ is given in the following table[2]:

ω	$P^*(\omega)$	ω	$P^*(\omega)$	ω	$P^*(\omega)$	ω	$P^*(\omega)$
abc	0.1950	$ab\overline{c}$	0.1758	$a\overline{b}c$	0.0408	$a\overline{b}\overline{c}$	0.0519
$\overline{a}bc$	0.1528	$\overline{a}b\overline{c}$	0.1378	$\overline{a}\overline{b}c$	0.1081	$\overline{a}\overline{b}\overline{c}$	0.1378

4.2 Loading an Epistemic State

Another way to initialize the system with an epistemic state is to load such a state directly from the environment (where it might have been stored during a previous run of the system; this could be modelled easily by an additional top-level function). Therefore, there is a monitored nullary function *stored_state* : *EpState* which is used in the following rule:

if $do = Load$
then $currstate := stored_state$
$do := undef$

4.3 Querying an Epistemic State

The function

$$belief: EpState \times \mathcal{P}(Sen_{\mathcal{U}}) \to \mathcal{P}(Sen_{\mathcal{Q}})$$

is the so-called belief measure function which is subject to the condition

$$belief(\Psi, \mathcal{S}) \;=\; \{S[x] \mid S \in \mathcal{S} \text{ and } \Psi \models_{\mathcal{Q}} S[x]\}$$

for every $\Psi \in EpState$ and $S \subseteq Sen_{\mathcal{U}}$. For a given state Ψ, the call $belief(\Psi, S)$ returns, in the form of measured sentences, the beliefs that hold with regard to the set of basic sentences $S \subseteq Sen_{\mathcal{U}}$. The monitored function $queries : \mathcal{P}(Sen_{\mathcal{U}})$ holds the set of sentences and is used in the rule:

if $do = Query$
then $believed_sentences := belief(currstate, queries)$
$do := undef$

Example 4. Suppose the current epistemic state is $currstate = MaxEnt(\mathcal{S})$ from Example 3 above, and our query is "*What is the probability that unmarried students are young?*", i.e. $queries = \{(b|ac)\}$. The system returns $belief(currstate, queries) = \{(b|ac)[0.8270]\}$, that is, unmarried students are supposed to be young with probability 0.8270.

[2] $MaxEnt(\mathcal{S})$ has been computed with the expert system shell SPIRIT [RKI97a, RKI97b]; cf. `http://www.fernuni-hagen.de/BWLOR/spirit.html`

4.4 Revision of Conditional Knowledge

Belief revision, the theory of dynamics of knowledge, has been mainly concerned with propositional beliefs for a long time. The most basic approach here is the *AGM-theory* presented in the seminal paper [AGM85] as a set of postulates outlining appropriate revision mechanisms in a propositional logical environment. This framework has been widened by Darwiche and Pearl [DP97a] for (qualitative) epistemic states and conditional beliefs. An even more general approach, unifying revision methods for quantitative and qualitative representations of epistemic states, is described in [KI01a]. The crucial meaning of conditionals as *revision policies* for belief revision processes is made clear by the so-called *Ramsey test* [Ram50], according to which a conditional $(B|A)$ is accepted in an epistemic state Ψ, iff revising Ψ by A yields belief in B:

$$\Psi \models (B|A) \quad \text{iff} \quad \Psi * A \models B \tag{2}$$

where $*$ is a belief revision operator (see e.g. [Ram50, BG93]).

Note, that the term "belief revision" is a bit ambiguous: On the one hand, it is used to denote quite generally *any* process of changing beliefs due to incoming new information [Gär88]. On a more sophisticated level, however, one distinguishes between different kinds of belief change. Here, *(genuine) revision* takes place when new information about a static world arrives, whereas *updating* tries to incorporate new information about a (possibly) evolving, changing world [KM91]. *Expansion* simply adds new knowledge to the current beliefs, in case that there are no conflicts between prior and new knowledge [Gär88]. *Focusing* [DP97b] means applying generic knowledge to the evidence present by choosing an appropriate context or reference class. *Contraction* [Gär88] and *erasure* [KM91] are operations inverse to revision and updating, respectively, and deal with the problem of how to "forget" knowledge. In this paper, we will make use of this richness of different operations, but only on a surface level, without going into details. The explanations given above will be enough for understanding the approach to be developed here. An interested reader may follow the mentioned references. For a more general approach to belief revision both in a symbolic and numerical framework, cf. [KI01a]. The revision operator $*$ used above is most properly looked upon as a *revision or updating* operator. We will stick, however, to the term *revision*, and will use it in its general meaning, if not explicitly stated otherwise.

The universe of revision operators is given by

$$\mathit{RevisionOp} = \{\mathit{Update}, \mathit{Revision}, \mathit{Expansion}, \mathit{Contraction}, \mathit{Erasure}, \mathit{Focusing}\}$$

and the general task of revising knowledge is realized by a function

$$\mathit{revise}: \ \mathit{EpState} \times \mathit{RevisionOp} \times \mathcal{P}(\mathit{Sen}_{\mathcal{Q}}) \rightarrow \mathit{EpState}$$

A call $\mathit{revise}(\Psi, \mathit{op}, \mathcal{S})$ yields a new state where Ψ is modified according to the revision operator op and the set of sentences $\mathcal{S}$. Note that we consider here belief

revision in a very general and advanced form: We revise epistemic states by sets of conditionals – this exceeds the classical AGM-theory by far which only deals with sets of propositional beliefs.

The constraints the function *revise* is expected to satisfy depend crucially on the kind of revision operator used in it and also on the chosen framework (ordinal or e.g. probabilistic). Therefore, we will merely state quite basic constraints here, which are in accordance with the AGM theory [AGM85] and its generalizations [DP97a, KI01a].

The first and most basic constraint corresponds to the *success postulate* in belief revision theory: if the change operator is one of *Update*, *Revision*, *Expansion*, the new information is expected to be present in the posterior epistemic state:

$$\begin{aligned} \mathit{revise}(\Psi, \mathit{Revision}, \mathcal{S}) &\models_{\mathcal{Q}} \mathcal{S} \\ \mathit{revise}(\Psi, \mathit{Update}, \mathcal{S}) &\models_{\mathcal{Q}} \mathcal{S} \\ \mathit{revise}(\Psi, \mathit{Expansion}, \mathcal{S}) &\models_{\mathcal{Q}} \mathcal{S} \end{aligned}$$

Furthermore, any revision process should satisfy *stability* – if the new information to be incorporated is already represented in the present epistemic state, then no change shall be made:

$$\begin{aligned} \text{If } \Psi \models_{\mathcal{Q}} \mathcal{S} \text{ then:} & \\ \mathit{revise}(\Psi, \mathit{Revision}, \mathcal{S}) &= \Psi \\ \mathit{revise}(\Psi, \mathit{Update}, \mathcal{S}) &= \Psi \\ \mathit{revise}(\Psi, \mathit{Expansion}, \mathcal{S}) &= \Psi \end{aligned}$$

Similarly, for the deletion of information we get:

$$\begin{aligned} \text{If not } \Psi \models_{\mathcal{Q}} \mathcal{S} \text{ then:} & \\ \mathit{revise}(\Psi, \mathit{Contraction}, \mathcal{S}) &= \Psi \\ \mathit{revise}(\Psi, \mathit{Erasure}, \mathcal{S}) &= \Psi \end{aligned}$$

To establish a connection between revising and retracting operations, one may further impose *recovery constraints*:

$$\begin{aligned} \mathit{revise}(\mathit{revise}(\Psi, \mathit{Contraction}, \mathcal{S}), \mathit{Revision}, \mathcal{S}) &= \Psi \\ \mathit{revise}(\mathit{revise}(\Psi, \mathit{Erasure}, \mathcal{S}), \mathit{Update}, \mathcal{S}) &= \Psi \end{aligned}$$

A correspondence between inductive knowledge representation and belief revison can be established by the condition

$$\mathit{inductive}(S) \;=\; \mathit{revise}(\mathit{uniform}, \mathit{Update}, S). \tag{3}$$

Thus, inductively completing the knowledge given by S can be taken as revising the non-knowledge representing epistemic state *uniform* by updating it to S.

In CondorASM, revision is realized by the rule

if $do = Revision$
then $currstate := revise(currstate, rev_op, new_information)$
$do := undef$

where the monitored functions $rev_op : RevisionOp$ and $new_information : \mathcal{P}(Sen_{\mathcal{Q}})$ provide the type of revision operator to be applied and the set of new sentences to be taken into account, respectively.

Example 5. In a probabilistic framework, a powerful tool to revise (more appropriately: update) probability distributions by sets of probabilistic conditionals is provided by the *principle of minimum cross-entropy* which generalizes the principle of maximum entropy in the sense of (3): Given a (prior) distribution, P, and a set, $\mathcal{S}$, of probabilistic conditionals, The *MinEnt-distribution* $P_{ME} = MinEnt(P, \mathcal{S})$ is the (unique) distribution that satisfies all constraints in $\mathcal{S}$ and has minimal cross-entropy with respect to P, i.e. P_{ME} solves the minimization problem

$$\min R(P', P) = \sum_{\omega} P'(\omega) \log \frac{P'(\omega)}{P(\omega)} \tag{4}$$
$$\text{s.t. } P' \text{ is a probability distribution with } P' \models \mathcal{S}$$

If $\mathcal{S}$ is basically compatible with P (i.e. P-*consistent*, cf. [KI01a]), then P_{ME} is guaranteed to exist (for further information and lots of examples, see [Csi75, PV92, Par94, KI01a]). The cross-entropy between two distributions can be taken as a directed (i.e. asymmetric) information distance [Sho86] between these two distributions. So, following the principle of minimum cross-entropy means to revise the prior epistemic state P in such a way as to obtain a new distribution which satisfies all conditionals in $\mathcal{S}$ and is as close to P as possible. So, the *MinEnt*-principle yields a probabilistic belief revision operator, $*_{ME}$, associating to each probability distribution P and each P-consistent set $\mathcal{S}$ of probabilistic conditionals a revised distribution $P_{ME} = P *_{ME} \mathcal{S}$ in which $\mathcal{S}$ holds.

Example 6. Suppose that some months later, the agent from Example 3 has changed his mind concerning his formerly held conditional belief $(young|student)$ – he now believes that *students* are *young* with a probability of 0.9. So an updating operation has to modify P^* appropriately. We use *MinEnt*-revision to compute $P^{**} = revise(P^*, Update, \{(b|a)[0.9]\})$. The result is shown in the table below.

ω	$P^{**}(\omega)$	ω	$P^{**}(\omega)$	ω	$P^{**}(\omega)$	ω	$P^{**}(\omega)$
abc	0.2151	$ab\overline{c}$	0.1939	$a\overline{b}c$	0.0200	$a\overline{b}\overline{c}$	0.0255
$\overline{a}bc$	0.1554	$\overline{a}b\overline{c}$	0.1401	$\overline{a}\overline{b}c$	0.1099	$\overline{a}\overline{b}\overline{c}$	0.1401

It is easily checked that indeed, $P^{**}(b|a) = 0.9$ (only approximately, due to rounding errors).

4.5 Diagnosis

Having introduced these first abstract functions for belief revision, we are already able to introduce additional functions. As an illustration, consider the function

$$diagnose: \ EpState \times \mathcal{P}(Fact_{\mathcal{Q}}) \times \mathcal{P}(SimpleFact_{\mathcal{U}}) \to \mathcal{P}(SimpleFact_{\mathcal{Q}})$$

asking about the status of certain simple facts $D \subseteq SimpleFact_{\mathcal{U}} = \Sigma$ in a state Ψ under the condition that some particular factual knowledge $\mathcal{S}$ (so-called evidential knowledge) is given. It is defined by

$$diagnose(\Psi, \mathcal{S}, D) = belief(revise(\Psi, Focusing, \mathcal{S}), D)$$

Thus, making a diagnosis in the light of some given evidence corresponds to what is believed in the state obtained by adapting the current state by focusing on the given evidence.

Diagnosis is realized by the rule

if $do = Diagnosis$
then $believed_sentences := diagnose(currstate, evidence, diagnoses)$
$do := undef$

where the monitored functions $evidence : \mathcal{P}(Fact_{\mathcal{Q}})$ and $diagnoses : \mathcal{P}(SimpleFact_{\mathcal{U}})$ provide the factual evidence and a set of (unquantified) facts for which a degree of belief is to be determined.

Example 7. In a probabilistic framework, focusing on a certain evidence is usually done by conditioning the present probability distribution correspondingly. For instance, if there is certain evidence for being a student and being unmarried – i.e. $evidence = \{student \wedge unmarried[1]\}$ – and we ask for the degree of belief of being young – i.e. $diagnoses = \{young\}$ – for $currstate = P^*$ from Example 3, the system computes

$$diagnose(P^*, \{student \wedge unmarried[1]\}, \{young\}) = \{young[0.8270]\}$$

and update *believed_sentences* to this set. Thus, if there is certain evidence for being an unmarried student, then the degree of belief for being young is 0.8270.

4.6 What-If-Analysis: Hypothetical Reasoning

There is a close relationship between belief revision and generalized nonmonotonic reasoning described by

$$\mathcal{R} \mid\!\sim_{\Psi} \mathcal{S} \quad \text{iff} \quad \Psi * \mathcal{R} \models_{\mathcal{Q}} \mathcal{S}$$

(cf. [KI01a]). In this formula, the operator $*$ may be a revision or an update operator. Here, we will use updating as the operation to study default consequences. So, *hypothetical reasoning* carried out by the function

$$nmr_{upd}: \ EpState \times \mathcal{P}(Sen_{\mathcal{Q}}) \times \mathcal{P}(Sen_{\mathcal{U}}) \to \mathcal{P}(Sen_{\mathcal{Q}})$$

can be defined by combining the *belief*-function and the *revise*-function:

$$nmr_{upd}(\Psi, \mathcal{S}_q, \mathcal{S}_u) \;=\; belief(revise(\Psi, Update, \mathcal{S}_q), \mathcal{S}_u)$$

The assumptions $\mathcal{S}_q$ used for hypotherical reasoning are being hold in the monitored function $assumptions : \mathcal{P}(Sen_{\mathcal{Q}})$ and the sentences $\mathcal{S}_u$ used as goals for which we ask for the degree of belief are being hold in the monitored function $goals : \mathcal{P}(Sen_{\mathcal{U}})$. Thus, we obtain the rule

if $do = What\text{-}If\text{-}Analysis$
then $believed_sentences := nmr_{upd}(currstate, assumptions, goals)$
$do := undef$

Example 8. With this function, hypothetical reasoning can be done as is illustrated e.g. by "Given P^* in Example 3 as present epistemic state – i.e. $currstate = P^*$ –, what would be the probability of $(b|c)$ – i.e. $goals = \{(b|c)\}$ –, provided that the probability of $(b|a)$ changed to 0.9 – i.e. $assumptions = \{(b|a)[0.9]\}$?" CONDOR's answer is $believed_sentences = \{(b|c)[0.7404]\}$ which corresponds to the probability given by P^{**} from Example 6.

4.7 Conditional Knowledge Discovery

Conditional knowledge discovery is modelled by a function

$$CKD : \; EpState \rightarrow \mathcal{P}(Sen_{\mathcal{Q}})$$

that extracts measured facts and rules from a given state Ψ that hold in that state, i.e. $\Psi \models_{\mathcal{Q}} CKD(\Psi)$. More significantly, $CKD(\Psi)$ should be a set of "interesting" facts and rules, satisfying e.g. some minimality requirement. In the ideal case, $CKD(\Psi)$ yields a set of measured sentences that has Ψ as its "designated" representation via an inductive representation formalism *inductive*. Therefore, discovering most relevant relationships in the formal representation of an epistemic state may be taken as solving the *inverse representation problem* (cf. [KI00, BKI02b]):

> Given an epistemic state Ψ find a set of (relevant) sentences S that has Ψ as its designated representation, i.e. such that $inductive(S) = \Psi$.

The intended relationship between the two operations *inductive* and *CKD* can be formalized by the condition

$$inductive(CKD(\Psi)) \;=\; \Psi$$

which holds for all epistemic states Ψ.

This is the theoretical basis for our approach to knowledge discovery. In practice, however, usually the objects knowledge discovery techniques deal with are not epistemic states but statistical data. We presuppose here that these data are at hand as some kind of distribution, e.g. as a frequency distribution or

an ordinal distribution (for an approach to obtain possibility distributions from data, cf. [GK97]). These distributions will be of the same type as epistemic states in the corresponding framework, but since they are ontologically different, we prefer to introduce another term for the arguments of *CKD*-functions:

$$distribution : EpState$$

if $do = CKD$
then $believed_sentences := CKD(distribution)$
$do := undef$

For instance, in a quantitative setting, Ψ may be a (rather complex) full probabilistic distribution over a large set of propositional variables. On the other hand, $CKD(\Psi)$ should be a (relatively small) set of probabilistic facts and conditionals that can be used as a faithful representation of the relevant relationships inherent to Ψ, e.g. with respect to the *MaxEnt*-formalism (cf. Section 4.1). So, the inverse representation problem for $inductive = MaxEnt$ reads like this: Find a set $CKD(\Psi)$ such that Ψ is the uniquely determined probability distribution satisfying $\Psi = MaxEnt(CKD(\Psi))$.

We will illustrate the basic idea of how to solve this *inverse MaxEnt-problem* by continuing Example 3.

Example 9. The probability distribution we are going to investigate is P^* from Example 3. Starting with observing relationships between probabilities like

$$P^*(\overline{a}b\overline{c}) = P^*(\overline{a}\overline{b}\overline{c}),$$

$$\frac{P^*(abc)}{P^*(ab\overline{c})} = \frac{P^*(\overline{a}bc)}{P^*(\overline{a}b\overline{c})},$$

$$\frac{P^*(a\overline{b}c)}{P^*(a\overline{b}\overline{c})} = \frac{P^*(\overline{a}\overline{b}c)}{P^*(\overline{a}\overline{b}\overline{c})},$$

the procedure described in [KI01b] yields the set $\mathcal{S}_u = \{(b|a), (b|c)\}$ of unquantified (structural) conditionals not yet having assigned any probabilities to them. Associating the proper probabilities (which are directly computable from P^*) with these structural conditionals, we obtain

$$\mathcal{S} = CKD(P^*) = \{(b|a)[0.8],\ (b|c)[0.7]\}$$

as a *MaxEnt*-generating set for P^*, i.e. $P^* = MaxEnt(\mathcal{S})$. In other words, the probabilistic conditionals

$(young|student)[0.8]$,
$(young|unmarried)[0.7]$

that have been generated from P^* fully automatically, constitute a consise set of uncertain rules that faithfully represent the complete distribution P^* in an information-theoretically optimal way. So indeed, we arrived at the same set of conditionals we used to build up P^* in Example 3. Thus, in this case we have

$$CKD(MaxEnt(\mathcal{S})) = \mathcal{S}$$

But note, that in general, $CKD(P)$ will also contain redundant rules so that only

$$CKD(MaxEnt(\mathcal{S})) \supseteq \mathcal{S}$$

will hold.

5 Conclusions and Further Work

Starting from a bird's-eye view of the CONDOR system, currently being under construction within our CONDOR project, we developed a high-level ASM specification for a system that provides powerful methods and tools for managing knowledge represented by conditionals. Thereby, we were able to elaborate crucial interdependencies between different aspects of knowledge representation, knowledge discovery, and belief revision.

Whereas in this paper, we deliberately left the universe $\mathcal{Q}$ of quantitative and qualitative scales abstract, aiming at a broad applicability of our approach, in a further development step we will refine CONDORASM by distinguishing quantitative logic (such as probabilistic logic) and qualitative approaches (like ordinal conditional functions). In vertical refinement steps, we will elaborate the up to now still abstract functions like *belief* and *revise* by realizing them on lower-level data structures, following the ASM idea of stepwise refinement down to executable code.

Acknowledgements: We thank the anonymous referees of this paper for their helpful comments. The research reported here was partially supported by the DFG – Deutsche Forschungsgemeinschaft within the CONDOR-project under grant BE 1700/5-1.

References

[AGM85] C. E. Alchourrón, P. Gärdenfors, and P. Makinson. On the logic of theory change: Partial meet contraction and revision functions. *Journal of Symbolic Logic*, 50(2):510–530, 1985. 193, 194

[BDP92] S. Benferhat, D. Dubois, and H. Prade. Representing default rules in possibilistic logic. In *Proceedings 3th International Conference on Principles of Knowledge Representation and Reasoning KR'92*, pages 673–684, 1992. 188

[BG93] C. Boutilier and M. Goldszmidt. Revision by conditional beliefs. In *Proceedings 11th National Conference on Artificial Intelligence (AAAI'93)*, pages 649–654, Washington, DC., 1993. 193

[BKI02a] C. Beierle and G. Kern-Isberner. Footprints of conditionals. In *Festschrift in Honor of Jörg H. Siekmann*. Springer-Verlag, Berlin, 2002. (to appear). 191

[BKI02b] C. Beierle and G. Kern-Isberner. Knowledge discovery and the inverse representation problem. In *Proceedings of the 2002 International Conference on Information and Knowledge Engineering, IKE'02*, 2002. 197

[Csi75] I. Csiszár. I-divergence geometry of probability distributions and minimization problems. *Ann. Prob.*, 3:146–158, 1975. 195

[DP97a] A. Darwiche and J. Pearl. On the logic of iterated belief revision. *Artificial Intelligence*, 89:1–29, 1997. 193, 194

[DP97b] D. Dubois and H. Prade. Focusing vs. belief revision: A fundamental distinction when dealing with generic knowledge. In *Proceedings First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97*, pages 96–107, Berlin Heidelberg New York, 1997. Springer. 193

[Gär88] P. Gärdenfors. *Knowledge in Flux: Modeling the Dynamics of Epistemic States*. MIT Press, Cambridge, Mass., 1988. 193

[GK97] J. Gebhardt and R. Kruse. Background and perspectives of possibilistic graphical models. In *Proceedings First International Joint Conference on Qualitative and Quantitative Practical Reasoning, ECSQARU-FAPR'97,*, pages 108–121. Springer, 1997. 198

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. 189

[KI98] G. Kern-Isberner. Characterizing the principle of minimum cross-entropy within a conditional-logical framework. *Artificial Intelligence*, 98:169–208, 1998. 191

[KI00] G. Kern-Isberner. Solving the inverse representation problem. In *Proceedings 14th European Conference on Artificial Intelligence, ECAI'2000*, pages 581–585, Berlin, 2000. IOS Press. 197

[KI01a] G. Kern-Isberner. *Conditionals in nonmonotonic reasoning and belief revision*. Springer, Lecture Notes in Artificial Intelligence LNAI 2087, 2001. 189, 193, 194, 195, 196

[KI01b] G. Kern-Isberner. Discovering most informative rules from data. In *Proceedings International Conference on Intelligent Agents, Web Technologies and Internet Commerce, IAWTIC'2001*, 2001. 198

[KM91] H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In *Proceedings Second International Conference on Principles of Knowledge Representation and Reasoning, KR'91*, pages 387–394, San Mateo, Ca., 1991. Morgan Kaufmann. 193

[Par94] J. B. Paris. *The uncertain reasoner's companion – A mathematical perspective*. Cambridge University Press, 1994. 191, 195

[PV92] J. B. Paris and A. Vencovská. A method for updating that justifies minimum cross entropy. *International Journal of Approximate Reasoning*, 7:1–18, 1992. 195

[PV97] J. B. Paris and A. Vencovská. In defence of the maximum entropy inference process. *International Journal of Approximate Reasoning*, 17:77–103, 1997. 191

[Ram50] F. P. Ramsey. General propositions and causality. In R. B. Braithwaite, editor, *Foundations of Mathematics and other logical essays*, pages 237–257. Routledge and Kegan Paul, New York, 1950. 193

[RKI97a] W. Rödder and G. Kern-Isberner. Léa Sombé und entropie-optimale Informationsverarbeitung mit der Expertensystem-Shell SPIRIT. *OR Spektrum*, 19/3, 1997. 192

[RKI97b] W. Rödder and G. Kern-Isberner. Representation and extraction of information by probabilistic logic. *Information Systems*, 21(8):637–652, 1997. 192

[Sho86] J. E. Shore. Relative entropy, probabilistic inference and AI. In L. N. Kanal and J. F. Lemmer, editors, *Uncertainty in Artificial Intelligence*, pages 211–215. North-Holland, Amsterdam, 1986. 195

[Spo88] W. Spohn. Ordinal conditional functions: a dynamic theory of epistemic states. In W. L. Harper and B. Skyrms, editors, *Causation in Decision, Belief Change, and Statistics, II*, pages 105–134. Kluwer Academic Publishers, 1988. 188

[SSB01] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. 189

A Appendix: Universes, Functions, Constraints, and Transition Rules for CondorASM

A.1 Universes

universe	typical element	
Σ	a	simple fact
Ω	ω	complete conjunction over Σ
$\mathcal{Q}$	x	measure, degree of belief
$SimpleFact_{\mathcal{U}} = \Sigma$	a	(unquantified) simple fact
$Fact_{\mathcal{U}}$	A	(unquantified) fact
$Rule_{\mathcal{U}}$	$(B\|A)$	(unquantified) conditional, rule
$Sen_{\mathcal{U}} = Fact_{\mathcal{U}} \cup Rule_{\mathcal{U}}$		(unquantified) sentences
$SimpleFact_{\mathcal{Q}}$	$a[x]$	quantified simple fact
$Fact_{\mathcal{Q}}$	$A[x]$	quantified fact
$Rule_{\mathcal{Q}}$	$(B\|A)[x]$	quantified conditional, rule
$Sen_{\mathcal{Q}} = Fact_{\mathcal{Q}} \cup Rule_{\mathcal{Q}}$		quantified sentences
$EpState$	Ψ	epistemic state
$WhatToDo$		domain of actions to be performed
$RevisionOp$	op	domain of revision operators

A.2 Static Functions

$$\models_{\mathcal{Q}} : EpState \times Sen_{\mathcal{Q}} \to Bool$$

$$uniform : EpState$$

$$inductive : \mathcal{P}(Sen_{\mathcal{Q}}) \to EpState$$

$$belief : EpState \times \mathcal{P}(Sen_{\mathcal{U}}) \to \mathcal{P}(Sen_{\mathcal{Q}})$$

$$revise : EpState \times RevisionOp \times \mathcal{P}(Sen_{\mathcal{Q}}) \to EpState$$

$$diagnose : EpState \times \mathcal{P}(Fact_{\mathcal{Q}}) \times \mathcal{P}(\Sigma) \to \mathcal{P}(Sen_{\mathcal{Q}})$$

$$nmr_{upd} : EpState \times \mathcal{P}(Sen_{\mathcal{Q}}) \times \mathcal{P}(Sen_{\mathcal{U}}) \to \mathcal{P}(Sen_{\mathcal{Q}})$$

$$CKD : EpState \to \mathcal{P}(Sen_{\mathcal{Q}})$$

A.3 Dynamic Functions

1. **Controlled Functions**:

function	arity	
currstate	: *EpState*	current epistemic state
believed_sentences	: $\mathcal{P}(Sen_{\mathcal{Q}})$	believed sentences w.r.t. current action

2. **Monitored Functions**:

function	arity	
rule_base	: $\mathcal{P}(Sen_{\mathcal{Q}})$	set of rules for initialization
stored_state	: *EpState*	epistemic state for loading
queries	: $\mathcal{P}(Sen_{\mathcal{U}})$	unquantified sentences for querying
new_information	: $\mathcal{P}(Sen_{\mathcal{Q}})$	set of new rules for revision
rev_op	: *RevisionOp*	revison operator
evidence	: $\mathcal{P}(Fact_{\mathcal{Q}})$	evidence for diagnosis
diagnoses	: $\mathcal{P}(SimpleFact_{\mathcal{U}})$	possible diagnoses to be checked
assumptions	: $\mathcal{P}(Sen_{\mathcal{Q}})$	assumptions for hypoth. reasoning
goals	: $\mathcal{P}(Sen_{\mathcal{U}})$	goals for hypothetical reasoning
distribution	: *EpState*	distribution for conditional knowledge discovery

3. **Interaction Functions**:

function	arity	
do	: *WhatToDo*	current action to be performed

A.4 Constraints

$$\mathit{belief}(\Psi, \mathcal{S}) \;=\; \{S[x] \mid S \in \mathcal{S} \text{ and } \Psi \models_{\mathcal{Q}} S[x]\}$$

$$\mathit{uniform} \;=\; \mathit{inductive}(\emptyset)$$

$$\mathit{revise}(\Psi, \mathit{Revision}, \mathcal{S}) \;\models_{\mathcal{Q}}\; \mathcal{S}$$
$$\mathit{revise}(\Psi, \mathit{Update}, \mathcal{S}) \;\models_{\mathcal{Q}}\; \mathcal{S}$$
$$\mathit{revise}(\Psi, \mathit{Expansion}, \mathcal{S}) \;\models_{\mathcal{Q}}\; \mathcal{S}$$

If $\Psi \models_{\mathcal{Q}} \mathcal{S}$ then:

$$
\begin{aligned}
revise(\Psi, Revision, \mathcal{S}) &= \Psi \\
revise(\Psi, Update, \mathcal{S}) &= \Psi \\
revise(\Psi, Expansion, \mathcal{S}) &= \Psi
\end{aligned}
$$

If not $\Psi \models_{\mathcal{Q}} \mathcal{S}$ then:

$$
\begin{aligned}
revise(\Psi, Contraction, \mathcal{S}) &= \Psi \\
revise(\Psi, Erasure, \mathcal{S}) &= \Psi
\end{aligned}
$$

$$
\begin{aligned}
revise(revise(\Psi, Contraction, \mathcal{S}), Revision, \mathcal{S}) &= \Psi \\
revise(revise(\Psi, Erasure, \mathcal{S}), Update, \mathcal{S}) &= \Psi
\end{aligned}
$$

$$
\begin{aligned}
inductive(S) &= revise(uniform, Update, S). \\
diagnose(\Psi, \mathcal{S}, D) &= belief(revise(\Psi, Focusing, \mathcal{S}), D) \\
nmr_{upd}(\Psi, \mathcal{S}_q, \mathcal{S}_u) &= belief(revise(\Psi, Update, \mathcal{S}_q), \mathcal{S}_u) \\
inductive(CKD(\Psi)) &= \Psi
\end{aligned}
$$

A.5 Transition Rules

if $do = Initialization$
then $currstate := inductive(rule_base)$
$do := undef$

if $do = Load$
then $currstate := stored_state$
$do := undef$

if $do = Query$
then $believed_sentences := belief(currstate,\ queries)$
$do := undef$

if $do = Revision$
then $currstate := revise(currstate,\ rev_op,\ new_information)$
$do := undef$

if $do = Diagnosis$
then $believed_sentences := diagnose(currstate,\ evidence,\ diagnoses)$
$do := undef$

if $do = What\text{-}If\text{-}Analysis$
then $believed_sentences := nmr_{upd}(currstate,\ assumptions,\ goals)$
$do := undef$

if $do = CKD$
then $believed_sentences := CKD(distribution)$
$do := undef$

Formal Description of a Distributed Location Service for Mobile Ad Hoc Networks

András Benczúr[1], Uwe Glässer[2], and Tamás Lukovszki[3]

[1] Computer Science Research Institute of the Hungarian Academy of Sciences
benczur@sztaki.hu
[2] School of Computing Science, Simon Fraser University, BC, Canada
glaesser@cs.sfu.ca
[3] Heinz Nixdorf Institute, Paderborn University, Germany
tamas@hni.upb.de

Abstract. We define here a *distributed abstract state machine* (DASM) [7] of the network or routing layer of mobile ad hoc networks [13]. Such networks require routing strategies substantially different from those used in static communication networks, since storing and updating large routing tables at mobile hosts would congest the network with administration packets very fast. In [1], the *hypercubic location service* is presented, which considers a very strong definition of fault-tolerance thereby improving state-of-the-art ad hoc routing protocols in several respects. Our goal in modeling the protocols for the distributed location service and the position based routing is twofold. First, we support the definition and validation of wireless communication protocols and implementations based thereon. Second, we feel that the abstract computation model naturally reflects the *layering principle* of communication architectures in combination with an uncompromisingly local view of the application domain. Thus we can identify fundamental semantic concepts, such as concurrency, reactivity and asynchronism, directly with the related concepts as imposed by the given application context.

1 Introduction

In this paper, we define a *distributed abstract state machine* (DASM) model [7] of the network or routing layer protocol for mobile ad hoc networks. Such networks are designed for wireless communication and require no fixed infrastructure as the mobile hosts also perform routing tasks [13]. Thus, they are particularly suitable, for instance, for rescue operations in regions affected by a natural disaster. The dynamics of self-organizing mobile networks require routing strategies substantially different from those used in static communication networks. Storing and updating large routing tables at mobile hosts would congest the network with administration packets very fast.

Ad hoc routing received a lot of attention in recent years [13]. Several routing protocols have been developed. The so-called *proactive* protocols, like DSDV [15],

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 204-217, 2003.
© Springer-Verlag Berlin Heidelberg 2003

continuously maintain route information for all destinations, whereas the *reactive* ones construct the routes for the destinations as required, e.g. DSR [9] and TORA [14]. Neither type of those strategies is fully satisfying however: the proactive ones suffer from a congestion of administrative messages when devices begin to move relative fast; reactive strategies in contrast have the drawback of long network search routes and thus high initiation costs.

Li et al. [11] present a routing strategy which combines the advantages of proactive and reactive routing. Their *geographic location service* (GLS), for each mobile node, consists of a few other randomly selected nodes that act as its location servers. Based on the assumption that every node can determine its current geographic position at any time—e.g., by using a global positioning system or GPS—, the nodes periodically resend position information to the location servers by the same routing mechanism used for normal data transmission, e.g., simple geographic forwarding. To initiate communication, we find a location server of the destination node and obtain its geographic position. A location server can be determined by visiting a well defined sequence of nodes such that each node acts as a location server of the next node in the sequence.

In [1], the *hypercubic location service* is presented, which considers a very strong definition of fault-tolerance, thereby improving state-of-the-art ad hoc routing protocols in several respects. Our goal in modeling the protocols for the distributed location service and the position based routing is twofold. First, we support the definition and validation of wireless communication protocols and implementations based thereon. Second, we feel that the abstract computation model naturally reflects the *layering principle* of communication architectures [2] in combination with an uncompromisingly local view of the application. Thus we can identify fundamental semantic concepts, such as concurrency, reactivity and asynchronism, directly with the related concepts as imposed by the given application context.

The paper is organized as follows. Section 2 briefly outlines the characteristic features of the applied protocols and the abstract representation of the network topology for geographic ad hoc routing. Section 3 introduces the abstract data structures and computation model underlying our mathematical definition of the network model. The protocols for position based routing and the distributed location service are formally defined in Section 4 and Section 5 respectively. Section 6 concludes the paper.

2 Efficient Ad Hoc Routing

The routing protocol in [1] is divided into two sublayers, one for the location service and one for position based routing between known locations, as illustrated in Fig. 1. These two sublayers cooperatively form the network layer of a mobile ad hoc network. The location service is based on a highly fault tolerant, self-scaling architecture called *hypercubic location service* or HLS. For the position based routing sublayer, the *Yao-graph* is used, which contains approximate power minimal routing paths.

In order to describe the maintenance of the network and the routing protocol, the existence of a *Media Access* or MAC layer, e.g. IEEE 802.11, is assumed to resolve interference problems. Furthermore, it is assumed that the mobile hosts use directed radio so that they can increase the transmission power (in discrete steps) in a direction

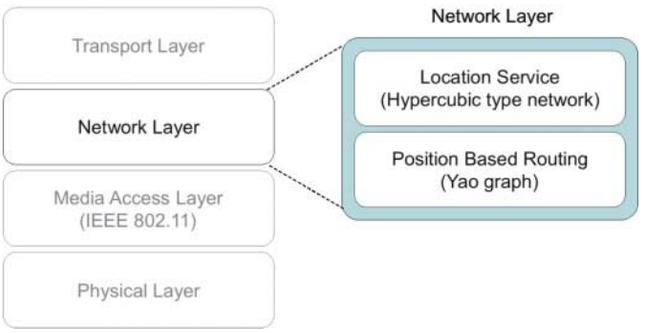


Fig. 1. Layered communication architecture

until at least one node eventually reacts by sending an acknowledge signal including its current geographic position.

2.1 Hypercubic Location Service

The hypercubic location service is presented in [1] and keeps most of the features of GLS [11], but also improves GLS in several ways. Here are the main aspects:

- It needs very little assumption on how the density varies across geographic subregions.
- It provides a method for very fast network topology recovery even after a failure of several mobile nodes—a task that requires reactive routing in GLS.
- It provides very strong fault tolerance, still keeping degrees as low as in the GLS.
- The network delivers a packet with a probability $1–n^{-c}$ even in the case when a constant fraction of the nodes crash at the same time.
- It survives a systematic destruction of devices within a region.
- It includes the self-scaling feature of the network.
- The location service forms a sublayer or the network layer, thus providing the flexibility of combining HLS with an arbitrary position based routing strategy.
- It keeps the administrative traffic close to that of [11] with a slight loss in the locality of communication, which can be fixed by a multi-layer solution.

Conceptually, HLS forms a sublayer of the network layer and consists of a dynamic and fault tolerant *hypercubic network* [10] with random IDs of the mobile devices identifying the nodes. Random ID distribution can be easily achieved by either hashing the node's IP or other physical address, or hard-wiring a random sequence into the device. The nodes are organized in a hypercubic network with respect to their random ID. In such networks neighbor IDs can be obtained by simple bit shift and flip operations of IDs. Each node in the network maintains location information of its hypercubic neighbors by periodically sending its own location to those neighbors. When the location of a node is known, a position based routing strategy can be used to pass messages to that node.

When a node u wants to send data to another node v, first u has to detect the location of node v. This happens as follows: The random ID of v can be obtained from the

random ID of u by performing a certain sequence of elementary bit operations. This sequence defines a path in the hypercubic network from u to v [10]. Node u sends a so-called *location request* for v on this path which contains the ID of u and v as well as the location of u.

Since the hypercubic neighbors are storing up-to-date location information from each other, the location request will be sent to the next node on the hypercube path using a certain position based routing. When the location request reaches v, then v sends its own location to u by using position based routing. After u has received the position of v it sends the data to v using the position based routing, as well.

As shown in [1], the random distribution of mobile node ID's ensures two key feature of the HLS:

- The location operations and the storage requirements are balanced across all the nodes.
- Since the IDs are independent of the geographic location or any other sensible network property, HLS resists faults of almost all nodes in a certain region, or a systematical demolition of a constant fraction of nodes.[1]

2.2 Position Based Routing

In order to route the message between two consecutive nodes of the hypercube path, any position based routing can be used, and, in particular, any appropriate geometric graph, such as spanner graphs [4], can be used as underlying network topology with any suitable routing strategy on that graph. In [1] the so-called *Yao-graph* [17] is proposed as network topology, because the following properties.

- Each node has a constant number of outgoing edges as illustrated in Fig. 2.
- The graph can be computed and maintained efficiently in a distributed manner (see [16] for example), allowing fast recovery from faults.
- It is a so-called power spanner [12] that allow low power routing and/or a position based routing where each node can determine the next node of the a path only knowing the geographic position of the destination node.

The Yao-graph $G_Y(V)=(V,E_Y)$ for a set V of n points in the plane is defined as follows (see Fig. 2a for an example). Let k be an integer and $\theta = 2\pi / k$. Let h_i, $i = 0,...,k-1$, be the half-line coincident with the rotated positive x-axis around the origo by an angle θ, and let c_i be the cone between h_i and $h_{i+1 \bmod k}$. For a point $p \in \mathbb{R}^2$ let $c_i(p)$, $i = 0,...,k-1$, be the translated cone c_i whose apex is at p. For each $p \in V$, let α_p be an angle, $0 \le \alpha_p < \theta$, and let $c_i'(p)$ be the rotated cone $c_i(p)$ around p by an angle α_p. We call the cones $c_i'(p)$ the *sectors* of p. For $p \in V$ and $c_i'(p)$, $i = 0,...,k-1$, let $n_i(p) \in S$

[1] The location service can be adapted to the required scale of fault tolerance. The more storage per node is allowed for storing geographic positions of other nodes the higher is the scale of fault tolerance that can be reached. HLS is planned for resisting even against the simultaneous fault of a constant fraction of the nodes. In that case a Packet will find the destination with a polynomial probability of $1-n^{-c}$ for a given constant c (see [1] for further details).

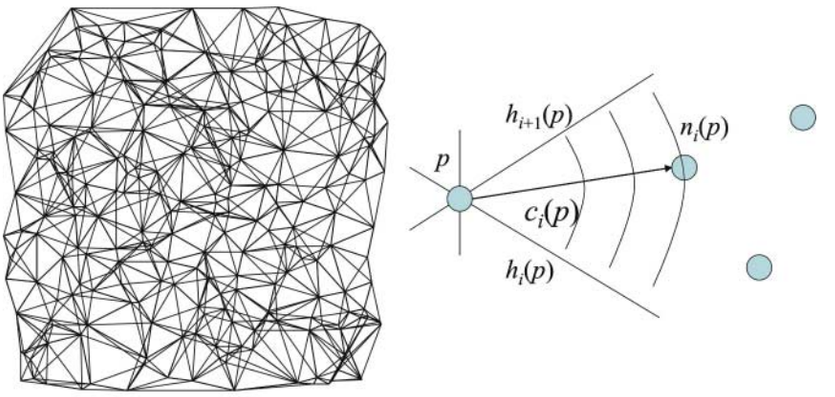


Fig. 2a. Yao graph for $k = 6$ with 256 nodes distributed uniformly at random within a square
Fig. 2b. Sector $c_i(p)$ with nearest neighbor $n_i(p)$

be the Euclidean nearest neighbor of p in $c_i'(p)$, if exists, and connect p to $n_i(p)$ by a directed edge (see Fig. 2b), i.e. $E_Y = \{ (p, n_i(p) : 0 \le i < k, c_i'(p) \cap V \neq \emptyset \}$.

We remark that the Yao-graph neighbors can be determined efficiently in a distributed and power efficient manner, as described in [16] for a variation of the Yao-graph. This distributed construction assumes that the mobile nodes use directed radio and they can increase the transmission power (in discrete steps) in the sectors until at least one node hears them and sends an acknowledge signal which also can contain its geographic position. Then the node only has to choose the closest one among the responding nodes in each sector. Here we assume for simplicity that each node is within the maximum transmission range of each other node.

3 Abstract Network Model

Consider a distributed communication architecture consisting of some finite number of mobile hosts interconnected through a wireless communication network. The global topology is formed by viewing the mobile hosts as *nodes* of the network. Each node has the ability to act as communication endpoint and as router at the same time. The network is ad hoc and as such changes its topology frequently and without prior notice. Nodes operate concurrently and interact with each other by asynchronously sending and receiving *packets* of variable length.

Conceptually, we distinguish two basically different views of communication, commonly referred to as the *horizontal* and the *vertical* view of communication. Any direct communication between peer protocol entities (i.e., entities residing within the same layer) at different nodes is called horizontal communication. This form of communication is only virtual and effectively realized by the lower level network layers. Therefore, the MAC layer renders a corresponding service to the network layer. This service is accessible through well defined interfaces, called *service access points*. Similarly, the network layer renders its service to the next higher layer, the transport

layer. Communication between different layers of the same node is called vertical communication. This form of communication is restricted to adjoining layers.

3.1 Concurrency and Real Time

We describe the concurrent and reactive behavior of network protocol entities by an asynchronous computation model in the form of a DASM. Autonomously operating protocol entities are *agents* interacting with each other, and also with the external world, by reading and writing shared locations of global machine states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [7].

Additionally, certain real time aspects need to be taken into account as well. Real time behavior imposes additional constraints on DASM runs ensuring that the agents react instantaneously [8]. To model the timing behavior faithfully, we introduce a notion of *local* system time in combination with mechanisms for handling timeout events. Our semantic model of time resembles those defined, e.g., in [3],[5],[8], except that we deliberately avoid a notion of global system time.

We construct our network model based on an intuitive level of understanding with a degree of detail and precision oriented towards practical needs in applied systems engineering. As such, our model serves as a basis for development and experimental validation. For a rigorous mathematical definition of the underlying computation model, we refer to the original literature on the theory of ASMs [7]. For a tutorial introduction to the DASM paradigm of systems modeling see also [6].

3.2 Communication Infrastructure

The total number N of nodes is a parameter of the communication network. Each node has a coordinate identifying its geographic *position* on the plane. As nodes may (and usually do) move, their positions change over time. The current position of a node is always known and the node can access this information locally through an interface to some location system, for instance, such as a global positioning system or GPS.

Domain NODE = {n_1, ..., n_N}

domain POSITION, monitored pos : NODE → POSITION

Every node has a unique ID or *address* within a fixed global address space. We assume a random distribution of the node IDs. This distribution is ensured by mapping some device specific address, e.g. such as Internet host names, IP addresses, or MAC addresses, to the node IDs using a strong hash function (not further specified here). Consider the resulting addresses as given by a static mapping from nodes to some abstract domain of addresses.[2]

domain ADDRESS, address : NODE → ADDRESS

[2] By not identifying nodes with their addresses, we gain additional flexibility allowing us, for instance, to model the dynamic assignment of addresses to nodes (e.g., as done in [5] for TCP/IP networks), although we do not exploit this option in the model as presented here.

We model the network layer by identifying the underlying protocol entities with corresponding DASM agents. Each node has two autonomously operating agents, one for the distributed location service and one for the position based routing. These agents execute the program *DistributedLocationService* and *PositionBasedRouting* as stated by a unary dynamic function *program* defined on agents. Thus, the domain *AGENT* is formed by the disjoint union of two sets of agents to which we refer as *DLS* and *PBR*. (Note that the function *program* is undefined on all those elements of the base set not representing agents.)

```
domain AGENT = DLS ∪ PBR, node : AGENT → NODE
```

In any given state of the network, the status of a node is either *switched on* (active), *switched off* (passive), or *undefined* (crashed). This status can change dynamically at run time depending on actions and events in the operational environment into which the network is embedded. For crashed nodes we assume that they do not interact with any other nodes. Consequently, they do not have any observable behavior and, as such, are not considered as agents of the DASM. Nonetheless, crashed nodes may recover becoming active or passive again. Recovery of crashed nodes is completely under the control of the external world and not further addressed here.

```
domain STATUS = { switched_on, switched_off }
```

```
monitored status : NODE → STATUS
```

For the representation of the local system time as associated with the individual mobile hosts, we introduce a monitored function *now* defined on nodes. Time values are given by a linearly ordered domain *TIME*. Intuitively, *now* represents the time as measured by some local clock. We do not state any constraints on the accuracy for measuring the local time with regard to the overall global system time.

```
domain TIME, now : NODE → TIME
```

3.3 Abstract Communication Model

Nodes send and receive packets consisting of two basically distinct parts, a packet *header* and the actual *payload*, as illustrated in Fig. 4. The header has a fixed structure identifying the receiver node, the sender node, and the packet type. For representing the information on the receiver and the sender within a packet header, we introduce a dynamic domain of abstract *node references*, called *NREF*.

```
domain PACKET
```

```
domain NREF, sndr,rcvr : PACKET → NREF
```

With every node reference we associate a node address and a geographic position.

```
address : NREF → ADDRESS, position : NREF → POSITION
```

With respect to the meaning that a packet has, one can distinguish three basically different types of packets, namely: (1) detection packets, which are meaningful for the position based routing only; (2) discovery packets, which are meaningful for the distributed location service only; and (3) data packets. For detection packets, we further distinguish between *neighbor requests* and *neighbor replies*. Similarly, for discovery packets, we distinguish between *location request*s and *location replies*.

```
domain DETECTION = {neighborRequest,neighborReply}
domain DISCOVERY = {locationRequest,locationReply}
type : PACKET → PACKETTYPE = DATAPACKET ∪ DETECTION ∪ DISCOVERY
```

We abstractly represent data as associated with a packet's payload using some not further specified domain *DATA*. Information encoded in the data part of detection packets and discovery packets is accessed by means of partial functions defined on packets (as will be explained). In the protocol description we do actually not consider the interpretation of data as associated with data packets any further.

```
domain DATA
```

According to the vertical view of communication, adjoining layers directly interact with each other through service access points. The network layer interacts with the MAC layer and the transport layer. Similarly, the position based routing, or *PBR*, interacts with the distributed location service, or *DLS*. Such interactions are restricted to the operations as provided by the *service primitives* of a service access point.

We use here two kinds of service primitives: (1) forwarding a single packet to the next high/lower layer; (2) receiving a single packet from the next high/lower layer.

```
PACKET_to_⟨Layer⟩(⟨Packet⟩[{,⟨Parameter⟩}*])
```

The tag ⟨*Layer*⟩ refers to an adjoining layer, ⟨*Packet*⟩ to the packet to be forwarded, and ⟨*Parameter*⟩ to additional information, such as the position of the destination node. We assume that the sender information is automatically attached to the packet header.

At the receiving layer, the above operation causes a *packet event* through which the receiving layer obtains the packet.

```
PACKET_from_⟨Layer⟩(⟨Packet⟩)
```

The tag ⟨*Layer*⟩ refers to the sending layer, and ⟨*Packet*⟩ to the packet being received. We formalize a packet event using a monitored Boolean-valued function. In any given state, the value of this function is *true* only in the presence of a packet event, and *false* otherwise. Whenever a packet event occurs, the formal parameter ⟨*Packet*⟩ is bound to an actual parameter identifying the packet being forwarded.

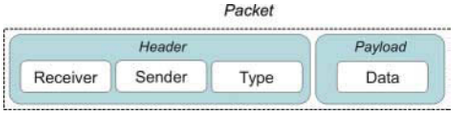


Fig. 3. Logical structure of packets for the communication between protocol entities

4 Position Based Routing

With every node we associate some fixed number k of *sectors,* where k is a parameter of the network. Directed communication within a given sector i, $0 \le i \le k-1$, is limited to those neighbors with a geographic position within the sector i. For every sector of a given node, the nearest (reachable) neighbor, if any, is identified locally through a dynamic mapping from node sectors to corresponding node references.

```
domain SECTOR = {0 .. k–1}
neighbor : NODE → (SECTOR → NREF)
```

Finite Euclidian distances on the plane are represented as positive real numbers from a linearly ordered domain *DISTANCE*. We introduce a binary operation *distance* on pairs of coordinates for calculating finite distances.

```
domain DISTANCE
distance : POSITION × POSITION → DISTANCE
```

4.1 Detection of Nearest Neighbors

The search for the nearest neighbors of a given node splits into k independent search operations, one for each of the k node sectors. That is, we must ensure that the search for a nearest neighbor within sector i does not interfere with the search for a nearest neighbor within another sector j, for $i \neq j$ and $1 \leq i,j < k$. Therefore, nearest neighbor request packets also specify the related sector ID so that the responses from neighbors then can use this ID to refer to a particular sector of the requesting node. Thus, every reachable neighbor can be uniquely assigned to one of the k sectors.

The sector information is encoded into the data part of the detection packets by means of a partial dynamic function *sector* defined on packets.

```
sector : PACKET → SECTOR
```

Nodes periodically update the information about their neighbors as the position of the nodes change dynamically and also because neighbors may become temporarily or permanently unreachable. The frequency of updates depends on various parameters, for instance, such as the relative speed of the neighbors and the node itself and the probability of node crashes. We abstract here from specific characteristics assuming the presence of externally controlled *update events* that trigger the start of detection cycles as indicated by the following function.[3]

```
monitored UpdateNeighborEvent : AGENT × SECTOR → BOOL
```

Intermediate results need to be stored temporarily until a detection cycle is completed. We therefore introduce an additional dynamic mapping from sectors to nodes.

```
newNeighbor : NODE → (SECTOR → NREF)
```

The detection of neighbors is a time critical operation since node positions may changes over time. Nevertheless, responses from neighbors may be subject to delays caused by retransmissions at lower layers and the fact that a neighbor may not always react instantaneously. Therefore, a timeout mechanism ensures that late responses are ignored. Every neighbor request also carries a *time stamp* encoded into the data part. The time stamp is then copied into the related reply. For accessing the time stamp, we introduce the following partial function on packets.

```
time : PACKET → TIME
```

---

[3] In any given state, the Boolean–valued function *UpdateNeighborEvent* has the value *false* on all those locations for which no detection cycle needs to be invoked.

Each sector of a node has its own timer operating under control of the PBR agent of that node. The time value of a timer specifies its expiration time. Initially, and after each reset operation, it has the distinguished value "∞" which is greater than all other time values. A timer with time value t, $now \leq t < \infty$, is considered to be active.

```
timer : AGENT × SECTOR → TIME ∪ {∞}
```

The actual time frame for detection cycles indeed is a parameter of the particular ad hoc network and depends on various application specific system characteristics. We represent this constant by a distinguished element *duration* from a static domain of finite time intervals that are given as positive real numbers.

```
domain DURATION, duration : DURATION
```

We can now define the rules for nearest neighbor detection that are executed by PBR agents as part of the *PositionBasedRouting* program defined in Sect. 0. In the below rules, *ag* refers to a DASM agent from *PBR, PBR* ⊂ *AGENT*.

```
NearestNeighborDetection=
  let node = ag.node, position = ag.node.pos, time = ag.node.now in
    forall s in SECTOR
      if UpdateNeighborEvent(ag,s) then // start new detection cycle
        IssueNeighborRequest(s,position,time)
        newNeighbor(node)(s):= undef
        timer(ag,s):= now(node) + duration // set timer
      if now(node) ≥ timer(ag,s) then // detection cycle timeout
        if newNeigbor(node)(s) ∈ NREF then // assign final result
          neighbor(node)(s):= newNeighbor(node)(s)
        timer(ag,s):= ∞ // reset timer
```

In order to generate a detection request, the position based routing creates a new detection request packet containing the relevant information, namely: the node address and geographical position, the sector ID, and a time stamp. By means of the MAC layer, this packet then is sent to nodes within the specified sector. For brevity, the formal definition of this operation is not given here.

On receiving a detection request, a detection reply packet is returned to the sender. The request and the reply are logically linked by copying the sector information and the time stamp of the request packet into the reply packet (see the below rule).

```
HandleNeighborRequest(p:PACKET) =
  extend PACKET with q
    extend NREF with r,s
      q.rcvr:= r
      q.sndr:= s
      r.address:= p.sndr.address, r.position:= p.sndr.position
      s.address:= ag.node.address, s.position:= ag.node.pos
      q.type:= neighborReply
      q.time:= p.time // copy time stamp from request packet
      q.sector:= p.sector // copy sector information from request packet

      Packet_to_MAC(q,p.sndr.position)
```

Finally, detection replies need to be handled as well. The selection of neighbors depends on the distance to a responding neighbor as well as on the distance to the nearest neighbor detected so far. This operation is specified by the below rule.

```
HandleNeighborReply(p:PACKET) =
  let s = p.sector, node = ag.node, pos = ag.node.pos
    if p.time ≥ ag.node.now − duration then
      // check distance against intermediately stored nearest neighbor
      let α = pos, β = p.sndr.position, γ = position(newNeigbor(node)(s))
        if distance(α,β) < distance(α,γ) then
          newNeigbor(node)(s):= p.sndr
```

4.2 Packet Routing

When receiving some packet from the MAC layer, the position based routing decides whether the packet can be delivered locally, or it needs to be forwarded to a remote destination. Delivering a packet locally means that the packet is handed over to the distributed location service running on the local node. Forwarding a packet means to send it to the nearest neighbor within the sector that matches with the position of the final destination node.

Depending on the position of the destination relative to the position of the local node, the respective nearest neighbor is computed by means of the following function.

```
computeNeighbor: NODE × POSITION → NREF
```

Below we define the model of the position based routing layer through the DASM program *PositionBasedRouting*. In addition to performing the actual routing task, this model also includes the control of nearest neighbor detection cycles and the handling of detection requests and detection replies. In the below program, *ag* refers to a DASM agent from *PBR* executing this program.

```
PositionBasedRouting=
  if status(ag.node) = switched_on
    NearestNeighborDetection // run detection cycle in parallel
    if Packet_from_MAC(p:PACKET) then
      if p.type = neighborRequest then
        HandleNeighborRequest(p)
      else
        if p.type = neighborReply then
          HandleNeighborReply(p)
        else // packets other than detection packets
          if ag.node.address = p.rcvr.address then
            Packet_to_DLS(p) // deliver packet locally
          else // forward packet to remote destination
            if computeNeighbor(ag.node,p.rcvr.position) ∈ NREF then
              let v = computeNeighbor(ag.node,p.rcvr.position)
                Packet_to_MAC(p,v.position) // perform next hop
    if Packet_from_DLS(p:PACKET) then
      Packet_to_MAC(p,p.sndr.position)
  else // node is passive
    skip
```

5 Distributed Location Service

With each node we associate a set of direct neighbors in the dynamic hypercube (see Sect. 0) as identified by a dynamic function *hypercubicNeighbors* on nodes.

```
hypercubicNeighbors : NODE → NREF-set
```

For a given node and a given destination address, a hypercubic neighbor on the way to the final destination is computed using the following dynamic function.

```
nextHypercubicNeighbor : NODE × ADDRESS → NREF
```

The definition of this function is based on the definition of another dynamic function, called *computeNextID,* which performs the actual address calculation on the dynamic hypercube. For brevity, we refer to [1] for the definition of this address calculation.

```
computeNextID : ADDRESS × ADDRESS → ADDRESS

nextHypercubicNeighbor(u,a) = v ∈ NREF: v ∈ hypercubicNeighbors(u)
                ∧ v.address = computeNextID(u.address,a)
```

Consider some sequence of consecutive packets received, one by one, from the transport layer. The first packet of every such sequence requires special treatment. This packet needs to be stored locally to first determine the position of the destination node before sending the packet. Accordingly, we assume that the first packet always can be recognized as such using a monitored Boolean-valued function on packets.

```
monitored Firstpacket : PACKET → BOOL, firstpacket : DLS → PACKET
```

A location reply to the location service always matches a pending location request. Thus, the contained position information of the sender is added to a waiting first packet, which then eventually can be sent via the position based routing.

When receiving a location request, the location service checks the address of the final destination. This address is encoded into the data part of the discovery packet. We therefore introduce a partial dynamic function *address* defined on packets.

```
address : PACKET → ADDRESS
```

If the final destination address does not match with the local node, we calculate the next hypercubic neighbor on the way to the final destination and forward the request.

```
HandleDiscoveryPacket(p:PACKET) =
  if p.type = locationReply then // extract position information
    ag.firstpacket.rcvr.position:= p.sndr.position
    Packet_to_PBR(ag.firstpacket)
  if p.type = locationRequest then
    if p.address = ag.node.addres then // generate location reply
      HandleLocationReply(p)
    else // compute next hypercubic neighbor and forward packet
      let v = nextHypercubicNeighbor(ag.node,p.address)
        p.rcvr:= v // specify next hop in the hypercube
        Packet_to_PBR(p)
```

We can now define the model of the distributed location service through the below DASM program *DistributedLocationService* (where *ag* refers to a *DLS* agent).

```
DistributedLocationService=
  if Packet_from_Transport(p:PACKET) then
    if Firstpacket(p) then // discover destination position first
      DiscoverDestinationPosition(p.rcvr.address)
      ag.firstpacket:= p // intermediately store first packet
    else // destination position is already known
      Packet_to_PBR(p),
      p.rcvr.position:= ag.firstpacket.rcvr.position
  if Packet_from_PBR(p:PACKET) then
    if p.type ∈ DISCOVERY then
      HandleDiscoveryPacket(p)
    else Packet_to_Transport(p)
DiscoverDestinationPosition(a:ADDRESS) =
  let v = nextHypercubicNeighbor(ag.node,a)
    extend PACKET with q // create location request packet
      extend NREF with r,s
        q.rcvr:= r, r.address:= v.address, r.position:= v.position
        q.sndr:= s, s.address:= ag.node.address, s.position:= ag.node.pos
        q.type:= locationRequest
        q.address:= a
      Packet_to_PBR(q)
```

6 Concluding Remarks

We present a DASM model of the network layer protocol for geographic ad hoc routing based on the hypercubic location service defined in [1]. Conceptually, the network layer splits into two separate sublayers, one for the location service and one for the position based routing. Our model captures the core functionality of both sublayers exposing the key properties of the network layer protocol. The particular focus is on the overall interoperability of protocol entities rather than on the details of their internal realization. We formalize the basic algorithmic aspects with a degree of detail and precision that goes far beyond the informal, pseudocode like description of fundamental operational aspects as presented in [1].

The abstract operational view allows us to represent a relatively complex protocol in a concise and coherent way so that one can analyze the resulting behavior. Not yet included is the reorganization of the dynamic hypercube as required for the elimination and insertion of nodes in order to ensure fault tolerance of the distributed location service. This is feature defined in [1], and we do not see any principle difficulty for extending our model accordingly. Also not included are the operations to be performed when a node is switched on or switched off.

Acknowledgements

Our sincere appreciation to Peter Päppinghaus, Siemens Munich, and Margus Veanes, Microsoft Research Redmond, for their fairly detailed and valuable comments on a preliminary version of this paper.

References

[1] Benczúr and T. Lukovszki. A Degree *O*(*log log n*) Fault Tolerant Distributed Location Service for Geographic Ad-Hoc Routing. Technical Report, TR-RI-02-231, Heinz Nixdorf Institute, Paderborn, Germany, June 2002

[2] D. E. Comer. Internetworking with TCP/IP, *Principles, Protocols, and Architectures*. Prentice Hall, 2000

[3] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis and A. Prinz. Formal Definition of SDL-2000. *Journal of Universal Computer Science*, 11 (7): 1025-1050, Springer Pub. Co., 2001

[4] J. Gao, L. J. Guibas, J. Hershberger, L. Zhang, and A. Zhu. Geometric Spanner for Routing in Mobile Networks. In *ACM Symposium on Mobile Ad Hoc Networking and Computing* (MOBIHOC'01), pages 45-55, 2001

[5] U. Glässer, Y. Gurevich and M. Veanes. High-Level Executable Specification of the Universal Plug and Play Architecture. In Proc. *35th Hawaii International Conference on System Sciences*, Software Technology, IEEE Press, 2002

[6] U. Glässer, Y. Gurevich and M. Veanes. An Abstract Communication Model. Microsoft Research Technical Report, MSR-TR-2002-55, May 2002

[7] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger (Ed.), *Specification and Validation Methods*, Oxford University Press, 1995, 9-36

[8] Y. Gurevich and J. Huggins. The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions. In H. Kleine Büning (Ed.), *Computer Science Logic*, LNCS 1092, 266-290, Springer, 1995

[9] D. B. Johnson and D. A. Malz: Dynamic Source Routing in Ad Hoc Wireless Networks. Mobile Computing, Vol.: 353, Kluwer Academic Publishers, 1996.

[10] T. Leighton. Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes. Morgan Kaufmann, 1992.

[11] J. Li et al. A Scalable Location Service for Geographic Ad Hoc Routing. ACM International Conference on Mobile Computing and Networking (MobiCom'00), 120-130, 2000.

[12] X.-Y. Li and P.-J. Wan and Y. Wang. Power Efficient and Sparse Spanner for Wireless Ad Hoc Networks. IEEE International Conference on Computer Communications and Networks (ICCCN'01), 2001.

[13] M. Mauve, J. Widmer and H. Hartenstein. A Survey on Position-Based Routing in Mobile Ad-Hoc Networks. IEEE Network, 15(6): 30--39, 2001

[14] V. D. Park and M. S. Corson. A Highly Adaptive Distributed Routing Algorithm for Mobile Wireless Networks. IEEE INFOCOM, 1405-1413, 1997

[15] E. Perkins and P. Bhagwar. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. ACM SIGCOMM'94, 234-244, 1994

[16] R. Wattenhofer et al. Distributed Topology Control for Power Efficient Operaion in Wireless Multihop Ad Hoc Networks, IEEE INFOCOM, 2001

[17] C. Yao. On Constructing Minimum Spanning Trees in k-Dimensional Spaces and Related Problems. SIAM Journal on Comp., 11(4): 721-736, 1982

Remarks on Turbo ASMs for Functional Equations and Recursion Schemes

Egon Börger[1] and Tommaso Bolognesi[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] CNR, ISTI, Pisa
bolognesi@ei.pi.cnr.it

Abstract. The question raised in [15] is answered how to naturally model widely used forms of recursion by abstract machines. We show that turbo ASMs as defined in [7] allow one to faithfully reflect the common intuitive single-agent understanding of recursion. The argument is illustrated by turbo ASMs for Mergesort and Quicksort. Using turbo ASMs for returning function values allows one to seamlessly integrate functional description and programming techniques into the high-level 'abstract programming' by state transforming ASM rules.

1 Introduction

This paper has been triggered by two recent publications, namely [15] and [3]. In the first paper Moschovakis claims that identifying algorithms with abstract machines 'does not square with our intuitions about algorithms and the way we interpret and apply results about them' and suggests that 'algorithms are *recursive definitions* while machines model *implementations*, a special kind of algorithms'. A concrete challenge is stated in the form of a question, namely 'If algorithms are machines, then which machine is the mergesort?' In the second paper Blass and Gurevich react to what they call 'a provocative article...casting doubt on the ASM thesis'. Their answer to the concrete challenge consists in describing 'the mergesort algorithm, on its natural level of abstraction, in terms of distributed abstract state machines', instantiating the more general scheme in [13] with the intention to support the ASM thesis ('Every algorithm can be expressed, on its natural level of abstraction, by an ASM').

Before going to explain the reasons why we are dissatisfied with this solution and to expose what we believe to be a simpler answer which is based upon basic mono-agent ASMs and better supports the ASM thesis, a double proviso should make clear that our explanation is a technical one, driven by the concern for a clean and transparent integration of well-established programming techniques into high-level system design by ASMs. As a consequence our explanation tries to avoid the difficult epistemological issues raised by Moschovakis, Blass and Gurevich. On the one side, what is 'the natural level of abstraction' of an algorithm, in this particular case of mergesort, is to a large extent in the eye of the beholder,

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 218–228, 2003.
© Springer-Verlag Berlin Heidelberg 2003

but our impression (and the feedback we got from asking around) is that rarely somebody thinks of a *distributed* computation when reasoning about or trying to explain or to implement common recursive algorithms like mergesort. In fact Blass and Gurevich themselves admit in [3, Sect.6] that the runs of distributed ASMs defined in [11], which their explanation of mergesort-like recursions crucially relies upon, are 'a long story', whereas our students learn to understand and use recursion quickly and rather early in their career. Blass and Gurevich base their explanation of recursion on the conversion of recursive ASMs to distributed ASMs defined in [13], a quite complex procedure they invoke to express the standard mathematical term evaluation mechanism for recursive equations:

> Converting a recursive ASM to a distributed ASM amounts to making explicit the creation of vassals, the waiting for the vassals' return values, and the use of these return values to continue the computation.

On the other side we do not want to enter the discussion whether algorithms 'are' recursive definitions, although certainly there are many examples most of us would call algorithms which however are not reducible in a natural way to recursive definitions.

Our answer to Moschovakis' question is two-fold. Formulated in general terms it says that functional languages, e.g. Haskell, or more precisely their interpreters, are well-known abstract machines for mergesort-like algorithms, since they offer a perfect match for defining and computing algorithms by such systems of recursive equations. In that sense Moschovakis' question has been answered already a long time ago when the techniques for implementing functional programming languages were developed. However one may argue that defining such interpreters has to deal with features which are typical for *implementing* functional languages, like passing subcomputation values, spawning or deleting subtasks, etc.—routine matters for implementors of functional languages and involving much more than a mathematical user of systems of recursive equations would like to see. We take this as a legitimate argument and therefore provide here also a more concrete answer by directly defining the machinery which underlies the way how recursive equations are used by humans to compute the value of the function in question, avoiding any spurious implementation related feature. We explicitly extract here the high-level *mathematical* machinery which is tacitly used for the standard functional calculations and without which the equations would not constitute the description of an algorithm (but in the best case a specification, as Blass and Gurevich point out). This machinery is easily defined as a simple mono-agent abstract state machine, namely an abstract form of well-known mechanisms for algorithmic term evaluation procedures. It avoids to invoke multiple agents and distributed computations, and directly reflects what the average computer scientist refers to when explaining how to use recursive equations.

For concrete comparison we illustrate the argument on the mergesort algorithm discussed in [15, 3] and on quicksort, but our definitions explain in full generality how functional programming concepts and methods can be modeled

faithfully by a special class of ASMs and how they can thereby be naturally integrated into the state based ASM framework. Since we want our argument to provide further concrete evidence for the above stated ASM thesis [12, 2], we take care to provide our model for the conceptual ingredients of functional programming at a more abstract level than that of the ASM engines AsmGofer [16] and AsmL [9]—which implement an ASM based framework enriched by functional and object-oriented programming features—and also of the implementation of procedure calls in XASM [1, 17].

2 Turbo ASMs

As has often been critically observed, the characteristic feature of basic ASMs—simultaneous execution of multiple atomic actions in a global state—comes at a price, namely the lack of direct support for practical composition and structuring principles. To make such features available as standard refinements of basic ASMs, in [7] a class of ASMs has been defined which offers as building blocks sequential composition **seq**, iteration **iterate**, and parameterized (possibly recursive) submachines. These so-called *Turbo ASMs* realize a black-box view of their component machines, hiding the internals of their (sub)computations by compressing them into one step (hence the name) and thus fit the synchronous parallelism of standard ASMs.

Purely iterative turbo ASMs, built by **seq** and **iterate** from function updates using as background functions only the initial functions of recursion theory, have been shown in [7, Sect.3.3] to compute arbitrary computable functions in a way which combines the advantages of Gödel-Herbrand style functional and of Turing style imperative programming. For parameterized submachines also *value returning ASMs* have been defined by extracting from the global result of a turbo submachine computation what one wants to consider as result value. A simple variation of that definition suffices to support the more general goal we want to reach here, namely to justify the smooth integration of stateless recursive functional programming into the state based synchronous parallel ASM framework by a natural subclass of value returning turbo ASMs. These machines are one-agent ASMs with multiple, synchronous parallel rules (or submachines) which express the usual procedure for evaluation of functional recursive equations and thus are our candidates for answering Moschovakis' question.

The following definition of *turbo submachine call* is taken unchanged from [7, Sect.4].

Definition 1. *Let* $R(x_1, \ldots, x_n) = body$ *be the declaration of a named turbo ASM rule* R*, let* $\mathfrak{A}$ *be a state. If* $[\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$ *is defined, then also* $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$ *is defined and its value is*

$$[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}.$$

By this definition every call $R(a)$ of a turbo submachine provides its global result in *one* step (if it terminates at all), namely by yielding the cumulative

update set of its entire computation (where in the case of overwriting due to sequential execution the last write wins).

To exploit this atomic view of turbo ASMs for returning values by machines which are supposed to compute functions of their input, it suffices to project that value out of the total computational effect $[\![R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}}$. For this purpose in [7, Sect.5.2] the usual notation $l \leftarrow R(a_1, \ldots, a_n)$ is adopted. We want the machine to store the result of R in location l, so that at the end of the R-computation the location l will contain the intended result value, in accordance with the expectation the programmer associates with this notation. R is a named rule as in the preceding definition in which a reserved 0-ary function **result** occurs with a placeholder role for storing the return value[1]. **result** represents the interface offered for communicating results from a rule execution to a location l which can be determined by the caller. Formally this comes up to replace **result** in the computation of R by l.

$$[\![l \leftarrow R(a_1, \ldots, a_n)]\!]^{\mathfrak{A}} = [\![body[l/\ \mathbf{result}, a_1/x_1, \ldots, a_n/x_n]]\!]^{\mathfrak{A}}$$

This definition should not be misunderstood as an invitation to consider value returning machine calls as a form of side-effect producing evaluation of 'terms' $R(t)$. On the contrary we believe that it is a precious methodological insight of Gurevich's ASM concept to only have main effects—updates which change the given state as defined by the rules—and to *abstract* from any other effect, rather than having them around as uncontrolled 'side' effects. As a consequence we believe it to be a pragmatically advisable system design principle not to blur the distinction between terms (which evaluate to objects) and rules (which evaluate to update sets triggering state changes), although the functional and λ-calculus tradition in theoretical computer science shows how one can make sense out of identifying objects and algorithms by turning everything into a function. Sticking to the convenient distinctions does not prevent us from having a mechanism to extract from a state some particular values one may wish to consider as distinguished objects coming out from subcomputations. In fact calling $R(t)$ as defined in [7] means that R is executed as a turbo ASM with parameter t and produces a complete state change, namely through its update set produced starting from the call state. The notation $l \leftarrow R(t)$ allows the designer to project out of that 'result' state the value of a specific location, namely the one named by l[2]. The turbo ASM notation $R(t)$ by itself does not provide any state hiding mechanism (though it can easily be enhanced to do so by standard modularization methods for rule import/export, (re)naming conventions for submachine signatures, local state concepts, etc., see e.g. [7]).

[1] Should a tuple of return values be desired, one would have finitely many such 0-ary functions $\mathbf{result}_j$ for $j = 1, \ldots, m$.

[2] In [14, pg.19] it is stated that "the idea of XASM is to generalize the original idea of Gurevich, resulting in a more practical specification and implementation tool" by introducing a new concept of "XASM call" that "leads to a design where every construct (including expressions and rules of Gurevich's ASMs) is denoted by both a value and an update set" (op.cit.pg.2). This throwing together of algorithms and

3 Modeling Recursion by Value Returning Turbo ASMs

There are many ways to explain the meaning of various forms of recursion. The turbo ASM submachine concept reviewed above abstractly mimics a standard imperative calling mechanism, which provides the key for laying in this section a particularly simple and rigorous foundation for the common intuitive understanding of recursion in terms of single-agent ASM computations. Furthermore by the atomicity of their black-box computations, turbo ASMs allow us to reflect exactly the machinery which underlies the common mathematical use of *functional* equations to evaluate functions defined by recursion, as we illustrate in this section for the Quicksort and Mergesort algorithms.

The update set produced by executing a turbo ASM call represents the total effect of executing the submachine in the call state (atomicity of the turbo ASM computation). Using the machine to return a value adds a form of *functional abstraction* from everything in that computation except the resulting input-output (argument-value) relation[3]. Technically we combine the turbo ASM notation for value returning machines with the **let**-construct, mimicking the use of activation records to store parameters as local variables. Since for each submachine call a dedicated placeholder is needed to record the result of a subcomputation, in the following definition we apply an external function *new* to the dynamic set FUN_0 of 0-ary dynamic functions ('write variables'). *new* is supposed to provide each time a completely fresh location, i.e. a location which has never been used before and is also not used for any other simultaneous call of *new*. This implies that if a machine is invoked within a same step simultaneously on different arguments $(a_1, ...a_n)$, *new* is assumed to provide as many result locations as there are invocations. For a detailed mathematical definition of this use of *new* see [10], [8, Ch.2.4.4][4].

the objects they compute, blurring the distinction between rules and terms, may have led to the thorough misunderstanding of turbo ASMs which transpires from the statement in op.cit.pg 2 that turbo ASMs are "excluding the essential feature of both Anlauff's and May's original call to allow returning not only update sets, but as well a value." The definition from [7] reported above makes clear that the statement is simply wrong, as is the conclusion drawn from it in op.cit. pg.2 that "This restriction makes their (i.e. Börger's and Schmid's) call useless for the modeling of recursive algorithms" (see the next section). The reader is the judge whether the above definition of turbo ASMs, and their use to extract return values from subcomputations, reflects current programming practice and does it in a transparent way. As reported in the comment on May's work in [6], it was the desire to simplify—read: to avoid the complexity of unnecessarily general logico-algebraic concepts for explaining standard programming concepts—which led Börger and Schmid to look for an *elementary* and transparently implementable definition of submachines which fits the characteristic synchronous parallelism of ASMs.

[3] The pure functional effect of course is achieved only if the submachine computation on the caller's side affects only the result location.

[4] As alternative for the use of *new* one can turn **result** into a monadic function which takes the list of parameters as arguments, so that the results of invokations with different arguments are stored in different locations.

Definition 2. *(Using return values in turbo ASMs). Let R_i, S be arbitrary turbo ASMs with formal parameter sequences x_i of R_i and parameters y_i of S . For corresponding actual parameter sequences a_i we define:*

$$
\begin{array}{l}
\textbf{let } \{y_1 = R_1(a_1), \ldots, y_n = R_n(a_n)\} \textbf{ in } S \equiv \\
\quad \textbf{let } l_1, \ldots, l_n = new(FUN_0) \textbf{ in} \\
\qquad \textbf{forall } 1 \leq i \leq n \textbf{ do } l_i \leftarrow R(a_i) \\
\qquad \textbf{seq} \\
\qquad \textbf{let } y_1 = l_1, \ldots, y_n = l_n \textbf{ in } S
\end{array}
$$

This definition allows one to explicitly capture the abstract machine which underlies the common mathematical evaluation procedure for functional expressions, including those defined by forms of recursion. The definition does not invite to mix terms (which are to be evaluated to objects) and machines (which have to be executed to obtain the desired state change). In fact for rules R and terms t the use of $R(t)$ is justified by our definition only as part of a call of R (i.e. in a place where a rule is expected) and in one of the two forms $l \leftarrow R(t)$ or **let** $x = R(t)$ **in** S. Passing (by value) the result returned by a turbo ASM captures the implicit storage of intermediate values by subterms during the evaluation of functional equations[5]. We illustrate this by the following turbo ASM definitions of Quicksort and of Mergesort which directly translate the usual recursive definition of the algorithms to provide as result a sorted version of any given list.

The computation suggested by the well-known recursive equations to quicksort L proceeds as follows: FIRST partition the *tail* of the list into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), THEN *concatenate* the results placing $head(L)$ between them. The fact that this description uses various auxiliary list and comparison operations is reflected by the appearance of corresponding auxiliary functions in the following turbo ASM.

$$
\begin{array}{l}
\textsc{Quicksort}(L) = \\
\quad \textbf{if} \mid L \mid \leq 1 \textbf{ then result} := L \textbf{ else} \\
\qquad \textbf{let} \\
\qquad\quad x = \textsc{Quicksort}(tail(L)_{<head(L)}) \\
\qquad\quad y = \textsc{Quicksort}(tail(L)_{\geq head(L)}) \\
\qquad \textbf{in result} := concatenate(x, head(L), y)
\end{array}
$$

In the usual verbal paraphrase reported above of the recursive equations for Quicksort there is no mention of agents, certainly no mention of multiple agents or of creation of 'vassals', not even of 'return'ing (by whom? from where?) values, but only of a certain order of some computation steps which explicitly appear as

[5] Introducing a stack discipline constitutes a further step towards an implementation. Since for each call we provide a new location, we can abstract from their ordering (e.g. in a stack one being on top of the other) and from their deletion (which remains for a refinement by a garbage collector, e.g. by a stack operation *pop*).

the 'principal' steps, namely to first compute some intermediate values for certain arguments—the result values of the two recursive subcomputations—and then 'to use...these...values to continue the computation' to produce the desired 'main' result. This is exactly what our formal ASM turbo machine definition mentions, abstracting from all the details of how these subcomputations are organized, as does the system of recursive equations.

On the other side, it seems to us that the definition of our QUICKSORT turbo ASM exactly mimics the usual recursive equations for the algorithm, and is as far from 'modeling an implementation' as the 'recursive definitions' are. Our translation of functional terms $f(s)$ using the construct **let** $x = F(s)$ applied to turbo ASM computations F directly captures the common procedure to evaluate in a certain order the subterms appearing in equations, to 'keep in mind' a certain number of them for a certain time until the moment comes where they are used simultaneously[6]. This mixture of ordering and independence is captured by the appropriate mixture of synchronous parallelism, sequentialization and turbo effect (atomic submachine view) provided by turbo ASMs.

One can argue whether computing $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ is part of the quicksort machine computation or not. The answer to the question depends on the level of abstraction at which one wants to define the machine. One can show that the following procedural refinement of list partitioning in Quicksort is correct and provides a correct 'implementation' (to use Moschovakis' term) of the more abstract machine above, though we would consider it still as an abstract machine.

Data Refinement of QUICKSORT. One can refine QUICKSORT to a control state [7]turbo ASM where the partitioning of L into $L_{<head(L)}$ and $L_{\geq head(L)}$ is computed using the following basic ASM PARTITION(l, h, p), working on the representation of lists as functions $L\colon [r, s] \to VAL$ from intervals of natural numbers to a set of values. When $r < s$, *Partition* is started with the search boundaries $l = r, h = s$ and the list head $pivot = L(r)$. It terminates when reaching $l = h$ with $L(l) = pivot$, all L-elements smaller than the pivot to the left of l, and all the others at l or to the right of l. Until reaching $l = h$, the partitioning procedure alternates between searching from above for list elements $L(h) \leq pivot$ and searching from below for list elements $L(l) \geq pivot$. When such an element is encountered and it is different from the element at the other current search boundary—one of them is the pivot—, then the boundary elements $L(l), L(h)$ are swapped and the search switches to the other boundary.

[6] This virtual intermediate storage of 'keeping in mind' intermediate results computed for subterms, typical for the functional handling of equations, is reflected in the definition **let** $x = R(a)$ **in** S by providing new locations where to temporarily keep the results of the subcomputations, to be used 'later' when executing S.

When $L(h) \leq pivot \leq L(l) \leq L(h)$ before $l = h$ is encountered (namely when pivot has multiple occurrences in the list), h can be decreased by one.

$\textsc{Partition}(l, h, pivot) =$
  if $L(h) > pivot$ **then** $h := h - 1$
  elseif $L(l) < pivot$ **then** $l := l + 1$
  elseif $L(l) > L(h)$ **then**
    $L(l) := L(h)$
    $L(h) := L(l)$
  elseif $l < h$ **then** $h := h - 1$

We are going now to illustrate our argument further by the Mergesort example, which is the one discussed by Moschovakis and Blass and Gurevich. The computation suggested by the usual recursive equations to mergesort a given list L consists in FIRST splitting it into a $LeftHalf(L)$ and a $RightHalf(L)$ (if there is something to split) and mergesorting these two sublists separately (independently of each other), THEN to *Merge* the two results by an auxiliary elementwise *Merge* operation. This is expressed by the following turbo ASM which besides two auxiliary functions *LeftHalf*, *RightHalf* comes with an external function *Merge* defined below as a submachine.

$\textsc{Mergesort}(L) =$
  if$\mid L \mid \leq 1$ **thenresult**$:= L$ **else**
    let
      $x = \textsc{Mergesort}(LeftHalf(L))$
      $y = \textsc{Mergesort}(RightHalf(L))$
    in result$:= Merge(x, y)$

Usually also *Merge* is defined by a recursion, suggesting the following computation scheme which is formalized by the turbo ASM below. If both lists are non-trivial, by a case distinction the smaller one of the two list heads is determined and placed as the first element of the *result* list, concatenating it with the result of a separate and independent *Merge* operation for the two lists remaining after having removed the chosen smaller head element. The ι-operator in $\iota x(P)$ denotes the unique x with property P (if there is such an x).

$\textsc{Merge}(L, L') =$
  if $L = \emptyset$ **or** $L' = \emptyset$ **then result**$:= \iota l(l \in \{L, L'\}$ **and** $l \neq \emptyset)$
  elseif $head(L) \leq head(L')$ **then**
    let $x = \textsc{Merge}(tail(L), L')$ **in result**$:= concatenate(head(L), x)$
  elseif $head(L') \leq head(L)$ **then**
    let $x = \textsc{Merge}(L, tail(L'))$ **in result**$:= concatenate(head(L'), x)$

[7] Control state ASMs defined in [4] constitute a subclass of basic ASMs, extending the classical Finite State Machines by allowing rules to be executed when passing from one to another internal ('control') state. They are extensively used in [8].

Data Refinement of MERGESORT **and** MERGE**.** One can show that the following definition of equivalence of corresponding locations provides a rule-wise step-by-step refinement to a model where lists are represented as functions $L\colon [l,h] \to VAL$ from intervals of natural numbers to a set of values. We leave it to the reader to decide whether the result of this refinement should be considered to be an abstract machine or an implementation.

- $L = \emptyset \equiv l > h$, $\mid L \mid \leq 1 \equiv l \geq h$
- $head(L) = L(l), tail(L) = L \setminus \{(l, L(l))\}$
- $LeftHalf\,(L) = L \restriction [l, half\,(l+h)]$, $RightHalf\,(L) = L \restriction [half\,(l+h)+1, h]$
  where $half\,(2x) = x, half\,(2x+1) = x+1$ (for example)
- $concatenate(v, L) = \{(l, v)\} \cup RightShift(L, 1)$
  where $RightShift(L, n) = \lambda x.L(x-n)$

In [15] also the issue of *outputting intermediate results* is mentioned, e.g. to compute the infinite sequence of primes. This can easily be incorporated by updates of the form **result**:=**result** $*newItem$ which extend the current result by a new item. Thus if the fixpoint of a computation is infinite, **result** will become a piecewise computed infinite sequence.

4 Conclusion

Result returning turbo ASMs make two ingredients of the evaluation of functional equations explicit which are taken for granted and—from the specification point of view luckily—abstracted away by the functional notation: the implicit storage of intermediate values by subterms and the independence or ordering of their computation and use in the evaluation procedure. If one wants to consider these abstract machines as 'mathematical models of computers'[15], then one should view also systems of recursive equations that way. Through our refinement definition (Def. 2) the concrete details of the machine computation are appropriately hidden, and at the same time provide a seamless connection to further implementation steps through unfolding the machine notation. The natural and tight correspondence between sets of recursive equations and these machines contributes to the pragmatic evidence accumulated over the years (see [6]) that ASMs can indeed be tailored to describe arbitrary algorithms on their natural level of abstraction, here the level of evaluating functional equations.

We pointed out above that our answer to Moschovakis' question is quite different from the one in [13, 3] where it is argued that 'recursive computations are to be viewed as a special case of distributed computation' in which each recursive call is executed by a newly created callee agent expected to return his result to the caller. The way we define the result of the turbo ASM call implies that from the caller's view it is returned immediately, directly reflecting the functional view which only uses the result in the given evaluation process and abstracts from how and by whom the result has been obtained. As a consequence our explanation of recursion does not need to invoke multiple agents, but it is compatible with using them for a distributed implementation, which is at

a more detailed level of modeling than the functional one. To view recursive computations as a special case of distributed computation implies to considerably lower the level of computational abstraction where the algorithm is analysed (or to increase the level of detailing if one prefers to look at it the other way round[8]). A related criticism of pushing the interpretation of the ASM thesis beyond its natural limits has been formulated in [5].

Acknowledgement

We thank the following colleagues for discussing and critically commenting upon a draft of this paper: W. Ahrendt, R. Banach, H. Langmaack, P. Päppinghaus, D. Rosenzweig, R. Stärk, W. Zimmermann and an anonymous referee.

References

[1] M. Anlauff. XASM — an extensible, component-based abstract state machines language. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69–90. Springer-Verlag, 2000. 220

[2] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 3, 2002. 220

[3] A. Blass and Y. Gurevich. Algorithms vs. machines. *Bulletin EATCS*, 2002. 218, 219, 226, 227

[4] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999. 225

[5] E. Börger. Computation and specification models. A comparative study. In P. Mosses, editor, *Proc. FLoC'02 Workshop Action Semantics and Related Semantic Frameworks*, BRICS Series. Department of Computer Science at University of Aarhus, 2002. 227

[6] E. Börger. The origins and the development of the ASM method for high level system design and analysis. *J. of Universal Computer Science*, 8(1):2–74, 2002. 222, 226

[7] E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000. 218, 220, 221, 222

[8] E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003. 222, 225

[9] Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at `http://research.microsoft.com/foundations/AsmL/`, 2001. 220

[8] In fact Blass and Gurevich wonder in [3, Sect.4] whether one 'shouldn't consider a broader class of implementations' than the one apparently Moschovakis is refering to. Once a recursive call has become a computation in a distributed context, many different possibilities to organize the subcomputations show up which cannot even be formulated in terms of the functional recursive equations.

[10] N. G. Fruja and R. F. Stärk. The hidden computation steps of turbo Abstract State Machines. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003*. Springer, 2003. 222

[11] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. 219

[12] Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. 220

[13] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. of Universal Computer Science*, 3(4):233–246, 1997. 218, 219, 226

[14] P. Kutter. The formal definition of Anlauff's eXtensible Abstract State Machine. TIK-Report 136, ETH Zürich, June 2002. 221

[15] Y. N. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited—2001 and beyond*. Springer, 2001. 218, 219, 226

[16] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`. 220

[17] X.ASM. Xasm Open Source. Web pages at `http://www.xasm.org/`, 2001. 220

Integrating UML Static and Dynamic Views and Formalizing the Interaction Mechanism of UML State Machines

Alessandra Cavarra[1], Elvinia Riccobene[2], and Patrizia Scandurra[2]

[1] Oxford University Computing Laboratory
Wolfson Building Parks Road, - OX1 3QD - Oxford, U.K.
Alessandra.Cavarra@comlab.ox.ac.uk
[2] Dipartimento di Matematica e Informatica, Università di Catania
V.le A. Doria, 6 - 95125 Catania, Italy
riccobene@dmi.unict.it

Abstract. In this paper we address the problem of integrating UML static and dynamic views, and different behavioral views. We tackle these problems by providing (a) a mapping of UML metamodel static and behavioral elements into ASMs and (b) a precise compositional semantics for UML state machines.
Structural model elements are translated into an ASM vocabulary as collections of domains and functions. The dynamic view is captured by multi-agent ASMs reflecting the behavior modeled by UML state machines.
The interaction among UML state machines is achieved by providing the semantics for *actions* and *events* and refining the ASM model in [4] to formalize objects communication, i.e. signals exchange and operation calls mechanism.

1 Introduction

Although UML is nowadays considered the new standard for model specification, visualization, construction and documentation, practitioners from industry still do not find it easy to use because of the lack of a well defined model design process and of a clear semantics of the different kinds of diagrams UML allows to draw.

UML is a bunch of notations without an effective integration and with loose semantics. This leads to a more apparent than real understanding of models, difficulty to perform rigorous analysis, validation, verification, integrity of models, and difficulty to develop tools supporting mechanical validation and verification.

This work comes as part of a formalization effort necessary to build a simulation tool [7] for a subset of the UML, namely class, object, state and interaction diagrams. In fact, although in [4] we provide a formal semantics for UML state machines, this proved to be not enough when one wants to reason about state machines in a more general context, or integrating state machines with other UML structural and behavioral diagrams.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 229–243, 2003.
© Springer-Verlag Berlin Heidelberg 2003

UML is a graphical notation used to describe the structure and the dynamics of a software system. The language is constituted of two different types of diagrams representing the two views of a system: the *static view*, modeling information that does not evolve with time, and the *dynamic view*, where the evolution of the components of the system is shown.

Even if many approaches exist in literature to define a precise semantics of static and behavioral diagrams [11], the problem of the integration between the static and the semantic views of UML is still an open problem, as well as the integration among behavioral views is still missing.

In this paper we address this problem by providing (a) a mapping of UML metamodel static and behavioral elements into ASMs and (b) a precise compositional semantics for state machines.

The "static" semantics of a given UML model represents the structural aspects of the model and essentially includes classes, relationships, attributes and operations. Moreover, this view encompasses the "static dependencies" captured by the metamodel and regarding only those meta-classes (including data types) and their relationships that directly represent the types of the entities specified by the given UML model at run-time. Such concepts concerning the static view of a system are translated into an ASM vocabulary as collections of domains and functions.

The "dynamic" semantics of a UML model describes the behavioral aspect of the model. The dynamic of a system can imply two different typologies of changes in the symbolic instance of the model: changes in the states of individual objects, caused by the triggering of events and the consequent execution of actions; and changes in the overall state of the system (due to e.g. creation and destruction of objects, the system clock, etc.).

In UML the behavior of an object is modeled by a state diagram; in our ASM model, objects will be formalized as ASM agents and their state diagrams as ASM modules associated to those agents. All the agents of the same "type" (i.e. formalizing objects of the same class) execute the same module, but each with its own execution space (the memory of the object itself) and event queue.

In order to handle the problem of the interaction among UML state machines, we provide a semantics for *actions* and *events* and refine the multi-agent ASM model in [4] in order to formalize objects communication, i.e. signals exchange and operation calls mechanism.

2 A Subset of the UML

Although UML provides a wide range of notations, to fully describe the functionalities of a system it is sufficient to use class diagrams and state diagrams. Class diagrams show collections of declarative elements in the system, and their relationships. State diagrams specify the life-cycle of objects of a class, describing the possible sequences of states and actions through which an object can go during its lifetime as a result of reacting to discrete events. Interactions among

objects are modeled by means of interaction diagrams which specify how and when messages are sent between objects to perform a task.

2.1 UML Class and Object Diagrams

A class diagram (see Fig. 2) describes the static structure of a system, consisting of a number of classes and their relationships. The application concepts are modeled in UML as classes each of which describes a set of objects that hold information and *communicate* to implement a behavior. The information they hold is modeled as attributes; the behavior they perform is modeled as operations. Structural relationships between objects of different classes are represented by *associations*. The definition of associations may be enhanced by a name, role names and cardinality (multiplicity).

An object diagram (see Fig. 3) is a graph of instances of the entities represented in a class diagram, namely objects and links. They model the actual configuration of the system at a certain time.

2.2 UML State Diagrams: Basic Concepts

State diagrams focus on the event-ordered behavior of an object, a feature which is specially useful in modeling reactive systems. A state diagram shows the event triggered flow of control due to transitions which lead from state to state, i.e. it describes the possible sequences of states and actions through which a model element can go during its lifetime as a result of reacting to discrete events. A state reflects a situation in the life of an object during which this object satisfies some condition, performs some action, or waits for some event. According to the UML meta-model [3], states can belong to one of the following categories: *simple states*, *composite states* (sequential, concurrent, submachine), *final*, and *pseudostates* (initial, history, stub, junction, synch).

Transitions are viewed in UML as relationships between two states indicating that an object in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain conditions are satisfied [2]. UML statecharts include *internal*, *external* and *completion* transitions.

The semantics of event processing in UML state machines is based on the *run to completion* (rtc) assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been terminated. Therefore, an event is never processed when the state machine is in some intermediate, unstable situation.

Events may be specified by a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where they are no more deferred or where they trigger a transition. Fig. 5 shows an example of state diagram.

3 Mapping UML Models into ASM

The basic idea here consists on mapping every UML model into a multi-agent ASM model.

The architecture of the UML is based on a four-layer metamodel structure, which consists of the following layers [3]:

- **Meta-metamodel**: This layer forms the foundation for the metamodeling architecture. The primary responsibility of this layer is to define the language for specifying a metamodel.
- **Metamodel**: A metamodel is an instance of a meta-metamodel. The primary responsibility of the metamodel layer is to define a language for specifying models. Examples of metaobjects in the metamodeling layer are: Class, Attribute, Operation.
- **Model**: A model is an instance of a metamodel. The primary responsibility of the model layer is to define a language that describes an information domain. Examples of objects in the modeling layer are: Stack, buffer, pop.
- **User Objects**: User objects (a.k.a. user data) are an instance of a model. The primary responsibility of the user objects layer is to describe a specific information domain.

Fig. 1 shows the schema adopted to map UML (meta) model elements into ASMs.

This mapping addresses the metamodel, model, and user objects layers. The UML notation defined at the metamodel level has been manually translated

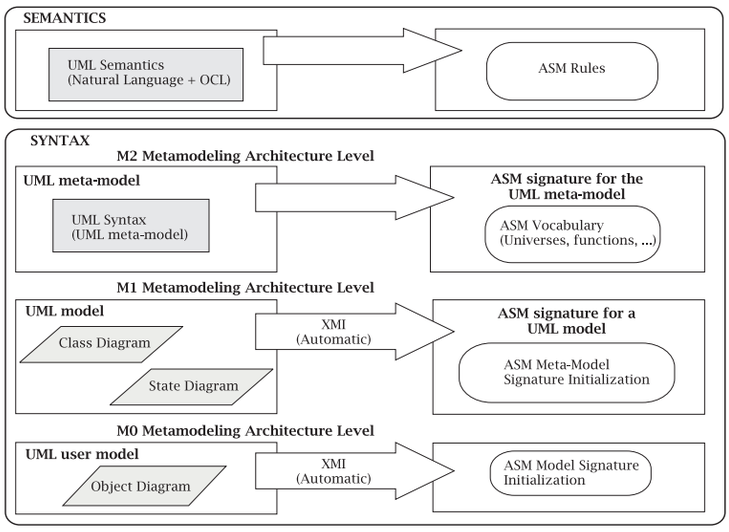


Fig. 1. UML to ASM mapping schema

once for all into a corresponding ASM signature, i.e. universes, functions, and predicates.

The mapping scheme has been implemented in our tool in order to automatically initialize the ASM signature from the input UML "model". In particular, class diagrams provide information regarding declarative UML elements (classes, operations, relationships,...). Object diagrams are used to define the multi-agent ASM model initial states. State diagrams allow to initialize the ASM signature formalizing the state machine elements (state, transitions, etc.). An ASM model formalizing the semantics of UML state diagrams has been defined (again once for all) according to the formal ASM model presented in [5, 4].

Observe that we assume input models to be notationally well defined.

3.1 Mapping UML Structural Elements into ASMs

We present here the translation process adopted to map basic UML structural elements, such as classes, relationships, attributes and operations into ASM.

As example of static mapping, we consider the UML model of the following stack-printer case study. A stack structure, of limited size, is an active object which is used by the environment to *push* and *pop* integer elements. The stack signals its state of underflow or overflow by sending a message of *stackEmpty* or *stackFull* to a printer. The printer is always ready to print all messages on a terminal. The terminal has a buffer *windowbuffer* of type String in which messages from the printer are showed.

Figures 2, 3, 5, 4 respectively show the class diagram, the object diagram, the state machine of the stack component, and the state machine of the printer component of the UML model of the stack-printer example, developed using the CASE tool Poseidon [10].

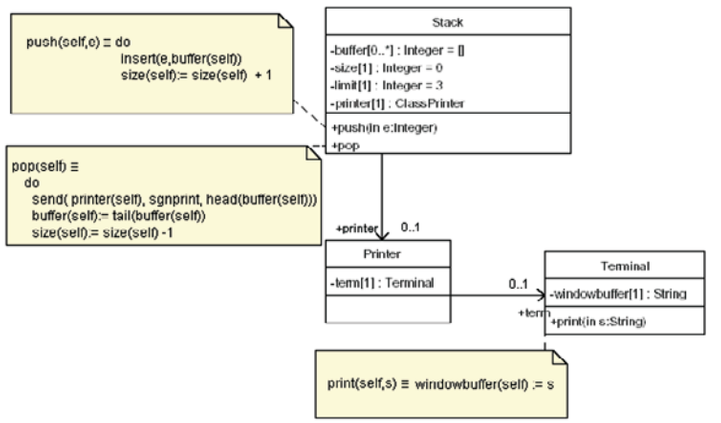


Fig. 2. Stack-Printer Class Diagram

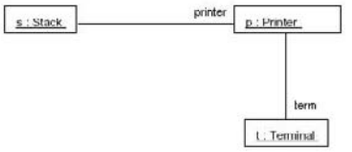


Fig. 3. Stack-Printer Object Diagram

Fig. 4. Printer State Machine

UML Metamodel. Each UML metaclass is mapped into a corresponding abstract set with the typewriting convention that a `MetaClass` is mapped into the ASM universe *MetaClass*, i.e. the metaclass `StateMachine` is mapped into the abstract set *StateMachine* of state machines, the metaclass `State` is mapped into the set *State*, `Transition` into *Transition*, `Event` into *Event*, `Action` into *Action*, etc.

A composition relation between metaclasses is mapped into a composition relation between the corresponding abstract sets. Therefore, the set *State* is partitioned into the sets *SimpleState*, *SequentialState*, *ConcurrentState*, *PseudoState*; the set *Event* is partitioned into the sets *SignalEvent*, *CallEvent*, *TimeEvent*, *ChangeEvent*; *Action* is partitioned into *CallAction*, *SendAction*, *CreateAction*, *DestroyAction*, *UninterpretedAction*; etc.

According to the UML Meta-Model for state machines, the ASM domain *StateMachine* has elements of type

$$statemachine(name, init, state, transition, event)$$

where $name \in String$ is the state diagram name, $init \in State$ is the initial state of the diagram, $state \in \mathcal{P}(State)$ denotes the set of states in the diagram, $transition \in \mathcal{P}(Transition)$ represents the set of (both internal and external) transitions in the diagram, and $event \in \mathcal{P}(Event)$ is the set of events labeling the transitions in the diagram.

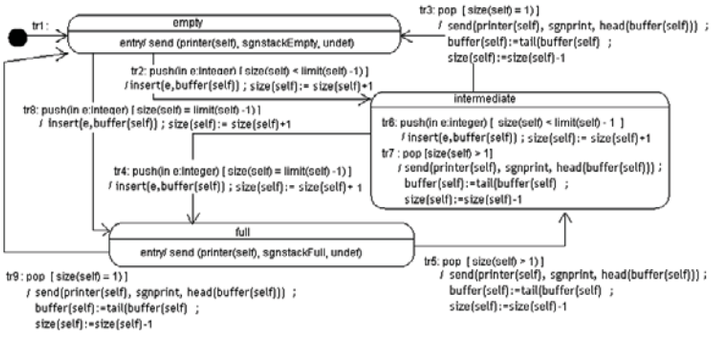


Fig. 5. Stack State Machine

UML Models. Information provided by the diagrams at the Model level, are used to initialize the abstract sets corresponding to the meta-classes at the Meta-Model level.

Let $\texttt{Class}_1,\ldots,\texttt{Class}_n$ the set given by the classes defined in all the class diagrams. For each $\texttt{Class}_i$, we define a *dynamic* domain $Class_i$ in the ASM model, whose elements are the instances of the corresponding class in the object diagrams (and will be initialized at User Model level).

Attributes are formalized as ASM functions, whose domain is the ASM universe corresponding to the class whom the attribute belongs to, and the co-domain is the attribute's type itself. *Operations* are represented by ASM rules parameterized on the universe corresponding to the class encompassing the operation and on the universes corresponding to the types of the operation parameters[1].

Each *association* navigable from $\texttt{Class}_i$ to $\texttt{Class}_j$ is represented as an ASM function having the universe $Class_i$ as domain and $Class_j$ as co-domain. If the association is navigable in both directions, then the inverse function is defined as well. A *generalization* between $\texttt{Class}_k$ and $\texttt{Class}_h$ is mapped into an inclusion relation between $Class_k$ and $Class_h$. Basic data types, such as Integer, String, Boolean, etc, are mapped into the corresponding ASM types. *Enumerations* will be translated as *static* ASM universes.

From class diagram in Fig.2, universes *Stack*, *Printer*, *Terminal*, *Integer*, *String* are defined (as elements of the abstract set *Class*) together with the functions:

$$
\begin{array}{l}
buffer,\ size,\ limit : Stack \rightarrow Integer^* \\
printer : Stack \rightarrow Printer \\
term : Printer \rightarrow Terminal \\
windowbuffer : Terminal \rightarrow String
\end{array}
$$

Rules for operations of the class `Stack` are the following:

$$
\begin{array}{l}
push(self,e) \equiv \textbf{do}\ insert(e,buffer(self)) \\
\qquad\qquad\quad\ \ size(self) := size(self) + 1
\end{array}
$$

$$
\begin{array}{l}
pop(self) \equiv \textbf{do}\ send(printer(self),sgnprint,head(buffer(self))) \\
\qquad\qquad\quad\ \ buffer(self) := tail(buffer(self)) \\
\qquad\qquad\quad\ \ size(self) := size(self) - 1
\end{array}
$$

where assignments are actions viewed as specializations of the metaclass `UninterpretedAction`. The operation of the class `Terminal` is defined by the rule:

$$
print(self,s) \equiv windowbuffer(self) := s
$$

Statecharts at Model level allow to initialize the abstract set *StateMachine* and the derived universes. From the state machines in Fig. 5 and 4, the universe *StateMachine* is initialized with the elements:

[1] We assume that operation parameters include possible return types.

$st1 = statemachine(StackMachine,\ initState1,\ \{empty,intermediate,\ full\},$
$\{tr1,\ tr2,\ tr3,\ tr4,\ tr5,\ tr8,\ tr9\} \cup \{tr6,tr7\},\ \{push(e{:}Integer),\ pop()\})$
$st2 = statemachine(PrinterMachine,\ initState2,\ \{active\},\ \{tr11\} \cup$
$\{tr22,\ tr33,\ tr44\},\ \{print(s{:}String),\ stackEmpty(),\ stackFull()\})$

UML User Model. Diagrams developed at User Model level (Object Diagrams) allow to initialize the signature defined at Model level.

Each *dynamic* domain $Class_i$ in the ASM model takes as elements all the instances of the corresponding class in the object diagrams. For instance, from the object diagram in Fig. 3, sets *Stack*, *Printer*, *Terminal* are initialized as $\{s\}$, $\{p\}$, $\{t\}$, respectively. And functions on $Class_i$ are suitably initialized ($buffer(s) = []$; $size(s) = 0$; $limit(s) = 3$; ... ; $term(p) = t$, $windowbuffer(p) =$ "", etc.).

Let *Object* be the union of all sets $Class_i$ defined as above. If the behavior of a class is described by a state diagram[2], the function $classifier : Object \rightarrow \{Class_1,\ldots,Class_n\}$ associates to each object the class it belongs to, while the function $context : StateMachine \rightarrow \{Class_1,\ldots,Class_n\}$ associates to a state machine its context, i.e. the set of objects behaving according to it;*behavior* is the inverse function of *context*. Each element in *StateMachine* is associated to a class instance which constitutes the context of the machine by the function $stMachine : Object \rightarrow StateMachine$ where $stMachine(a) = behavior(classifier(a))$. According to our example, $stMachine(s) = st1$ and $stMachine(p) = st2$.

3.2 Mapping UML Models Behavior into ASMs

In the UML models under consideration here, the behavior of class instances is described by a state diagram attached to the class. In our ASM model, objects with state machine are modeled as ASM agents (therefore, $AGENT \subseteq Object$)[3] and their state diagrams are formalized as ASM modules of such agents. All the agents (objects) of the same type (class) execute the same module (state diagram). However, each agent runs on its own execution space and handles its own event queue. The overall system behavior is represented by the multi-agent ASM whose initial states are given by the object diagrams.

According to our model of the Stack-Printer, $AGENT = \{s, p\}$, while t is a location on *Terminal*. The behavior of each agent is described by the state machine given by the function *stMachine*.

The ASM model for the behavioral meaning of state diagrams is defined in [5, 4]. This model (a) rigorously defines the UML event handling scheme in a way which makes all its "semantic variation points" explicit, including the event deferring and the event completion mechanism; (b) encapsulates the run to completion step in *two* simple rules (**Transition Selection** and **Generate Completion Events**) where the peculiarities relative to entry/exit or transition actions and sequential, concurrent or history states are dealt with in a modular

[2] It is possible to have classes without an associated state diagram.

[3] Since we do not distinguish between protocol and state machines, we associate to passive objects agents which are active *only* when stimulated by external events.

way; (c) integrates smoothly the state machine control structure with the data flow; (d) clarifies various difficulties concerning the scheduling scheme for internal ongoing (really concurrent) activities; (e) describes all the UML state machine features that break the thread-of-control; (f) provides a precise computational content to the UML terms of atomic and durative actions/activities, without loosing the intended generality of these concepts, and allows one to clarify some dark but semantically relevant points in the UML documents on state machines.

For each UML state machine in the model, the set of objects defined in the context of the diagram move through the diagram each executing what is required by their *current state* (see [5] for more details).

To fully understand the content of the following sections, a knowledge of the ASM state machines model in [4, 5] is strongly recommended.

3.3 ASM Semantics of UML Events

An event represents a specification of a noteworthy occurrence. In the UML metamodel four different kinds of event are defined: `SignalEvent`, `CallEvent`, `ChangeEvent`, and `TimeEvent` [3]. Currently, we allow only signal events and call events. A signal event represents the reception of a particular *asynchronous* signal. A call event represents the reception of a request to synchronously invoke a specific operation. The expected result is the execution of a sequence of actions which characterize the operation behavior at a particular state. We define the abstract sets *SignalEvent* and *CallEvent* to model the corresponding classes in the metamodel.

In UML it is assumed (*run-to-completion* (rtc) semantics) that a state machine processes one event at a time and finishes all the consequences of that event before processing another event. "An event is *received* when it is placed on the event queue of its target. An event is *dispatched* when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred as the *current event*. Finally, it is *consumed* when event processing is complete. A consumed event is no longer available for processing" [3].

Our model is fully compliant with the rtc assumption. The event handling mechanism is formalized by a number of queues, one per each agent (i.e. object with behavior) in the model. In such a way, we guarantee that each agent consumes an instance of an event at a time and also that distinct agents can *simultaneously* consume distinct instances of events. The function $eventQueue : AGENT \rightarrow Event*$ formalizes the association between an agent and its event queue.

According to the UML classification of events, the *eventQueue* of an agent may be updated at run-time by (a) the environment by means of *time events* (i.e. time expiration, calendar/clock, etc.), time *signal events* (i.e. stimuli from physical entities), (b) change in value of some agent entity by means of *change events*, (c) requests of services from other objects by means of *call events* (i.e. synchronous or asynchronous object operation invocations) and *signal events* (i.e. asynchronous signal reception). Therefore, we can imagine the *eventQueue*

of an agent as partitioned into *ExtEventQueue*, which is monitored by the environment and *IntEventQueue*, which is updated as result of *actions* of other agents (controlled by shared functions). Thus, for each agent a, $eventQueue(a) = ExtEventQueue(a) \cup IntEventQueue(a)$. The mechanism of updating *IntEventQueue* is explained in section 3.5 because related to the semantics of actions.

The ASM models for state machine in [5] and [4] have to be slightly refined in order to guarantee the concurrent computation of more than one machine at the same time. The monitored function $dispatched(e)$, indicating which event instance is dequeued to be processed, is refined with the predicate $dispatched(e, a)$, monitored on $eventQueue(a)$, $a \in AGENT$. In such a way, we allow different agents (even running the same type of machine) to process simultaneously instances of different events. An *eventQueue* is inherited by a subagent when created.

For each agent a, a transition is enabled to fire if it satisfies the following condition $enabled(t, e, a) \equiv event(t) = e \ \&\ guard(t)(a) \ \&\ source(t) \in currState(a)$. Let $enabled(e, a) = \{t \in transitions(stMachine(a)) \mid enabled(t, e, a)\}$ be the set of all transitions (simultaneously) enabled by an event e for an agent a.

Even if we allow more than one event *globally*, i.e. with respect to all the agents, dispatched at the same time, a single agent processes an event per time. However, more than one transition can be enabled by a same event in the state machine of the agent[4]. We skip here the mathematical definition of the set $FirableTrans(e, a)$ which consists of all triggering transitions taking into account the UML requirement that among all the transitions enabled by the same event and with different source states, priority is given to an innermost one and to completion transitions. The choice among those triggering transitions is left open as semantic variation point (see **choose** in the Transition Selection rule).

Deferred Events. Events may be specified in a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where they are no more deferred or where they trigger a transition. According to [3], an event that is deferred in a composite state is automatically deferred in all its directly or transitively nested substates. For reasons of simplicity, but without loss of generality, we assume that the defer set of each state explicitly contains all the inherited events to be deferred.

If a dispatched event does not trigger any transition in the current state of an agent, it is lost unless it occurs in the deferred set of the deepest active state. This is formalized by the following predicate *deferrable* on $Event \times AGENT$: $deferrable(e, a) = true \Leftrightarrow enabled(e, a) = \emptyset \ \&\ e \in defer(deepest(a))$.

As suggested in [3], to store deferred events we associate to each agent a list of events *deferQueue* that is dynamically updated during the computation (see rule Transition Selection). We can therefore define $deferred(e, a)$ to mean $e \in deferQueue(a)$.

[4] We refer the reader to [4] for a complete and exhaustive discussion on conflicting transitions.

We call a deferred event *releasable* when it becomes ready to be consumed, i.e. when it can trigger a transition in the current state configuration: $releasable(e, a) = true \Leftrightarrow deferred(e, a)\ \&\ enabled(e, a) \neq \emptyset$.

3.4 The *Run To Completion* Step

A run-to-completion-step of a state machine consists in choosing an event with an enabled transition, and in firing the enabled transition. Apparently, UML leaves it unspecified how to choose between dispatched and releasable events. We reflect this by using a selection function which, at any moment, chooses either a dispatched event triggering a transition, or an event that has been deferred. A dispatched event, if *deferrable*, has to be inserted into the *deferQueue*. A releasable event, when chosen for execution, has to be deleted from *deferQueue*. This implies that when choosing an event which is simultaneously *dispatched* and *releasable*, that event will be deleted from the deferred events.

The main rule for selecting the machine transition to be executed next can thus be stated as follows. For each $a \in AGENT$:

Rule **Transition Selection**(**Self**)
choose e : *dispatched*(*e,Self*) $\vee$ *releasable*(*e,Self*)
  choose *trans* **in** *FirableTrans*($e, Self$)
    stateMachineExecution(*trans*)
  if *deferrable*(*e,Self*) **then** *insert*(*e,deferQueue*(*Self*))
  if *releasable*(*e,Self*) **then** *delete*(*e,deferQueue*(*Self*))

The submachine *stateMachineExecution* is responsible for the execution of transitions.

State Machine Execution If an internal transition is triggered, then the corresponding action is executed (there is no change of state and no exit or entry actions must be performed). Otherwise, if an external transition is triggered, we must determine the correct sequence of exit and entry actions to be executed according to the transition source and target state. Transitions outgoing from composite states are inherited from their substates so that a state may be exited because a transition fires that departs from some of its enclosing states. If a transition crosses several state boundaries, several exit and entry actions may be executed in the given order. To this purpose, we seek the innermost composite state that encloses both the source and the target state, i.e. their *least common ancestor*. Then the following actions are executed sequentially: (a) the exit actions of the source state and of any enclosing state up to the direct substate of the least common ancestor (*ToS*), innermost first (see macro *exitState* in [4]); (b) the sequence of actions on the transition; (c) the entry actions of the target state and of any enclosing state up the direct substate of the least common ancestor (*FromS*), outermost first (see macro *entryState* in [4]); finally (d) the "nature" of the target state is checked and the corresponding operations are performed. For a precise, mathematical definition of the parameters *ToS* and *FromS* refer to [5].

Moreover, if the selected event is a *call event*, processing such event results in executing the *method* (if one exists) in the full descriptor of the class associated with the operation of the call event, and then the action sequence (if any) before entering the target state. This form of behavior corresponds to mix transitions with explicit actions with *protocol transitions*[5] in order to get a hybrid model (the current OMG specification doesn't prohibit this because there may be some applications that might benefit from mixing). In case no method is associated to the operation, the ASM rule *method* has no effect.

stateMachineExecution(*trans*) ≡
if *internal*(*trans*) **then** *action*(*trans*)
else seq
    exitState(*source*(*trans*),*ToS*)
    if *isCallEvent*(*event*(*trans*)) **then** *method*(*event*(*trans*))
    action(*trans*)
    enterState(*FromS*,*target*(*trans*))
    case *target*(*trans*)
        SequentialState: *enterInitialState*(*target*(*trans*))
        ConcurrentState: *startConcurrComput*(*target*(*trans*))
        HistoryState: *restoreConfig*(*target*(*trans*))
    endcase

The ASM constructs for sequentialization and iteration defined in [6] provide the combination of black box –atomic step– view and the white box –durative– view which is needed in the definition of the parameterized macro *stateMachineExecution* to guarantee that when the ASM rule is executed, all the updates which occur in the macros stated inside are performed before the next event is dispatched or becomes releasable.

3.5 ASM Semantics of UML Actions

An action is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a message to an object or modifying a value of an attribute. In UML, the language of actions and events is left partially undefined: its final form is dictated by the *target language* of model –usually the programming language chosen for implementation. Currently, an OMG request for proposal for a "precise, software-independent action specification" has been issued. In our framework, we decided to adopt an ASM dialect as action language for our models.

In the UML metamodel, `Action` is modeled as an abstract class from which seven types of actions are defined by specialization: `SendAction`, `CallAction`,

[5] A *protocol state machine* for a class defines the order in which the operations of that class can be invoked. The behavior of each of these operations is defined by an associated method, rather than through action expressions on transitions. A transition in a protocol state machine has as trigger a call event that references an operation of the class, and an empty action sequence.

`CreateAction`, `TerminateAction`, `DestroyAction`, `ReturnAction`, `UninterpretedAction`. A send action is an action that results in the (asynchronous) sending of a signal. A call action is an action resulting in an invocation of an operation on an instance, i.e. a call event. A call action can be synchronous or asynchronous, indicating whether the operation is invoked synchronously or asynchronously. This is modeled by the boolean attribute `isAsynchronous` of the class `Action`. A create action is an action resulting in the creation of an instance of some classifier. A destroy action is an action resulting in the destruction of an object specified in the action. A terminate action results in self-destruction of an object. A return action is an action that results in returning a value to a caller.

The class `Action` is mapped into the abstract set *Action*, partitioned into *SendAction*, *CallAction*, *CreateAction*, *TerminateAction*, *DestroyAction*, *AssignmentAction*, *ReturnAction*[6]. The elements of each partition are identified by the following characteristic functions, respectively: *isSendAction*, *isCallAction*, *isCreateAction*, *isTerminateAction*, *isDestroyAction*, *isReturnAction* and *isAssignmentAction*. The predicate $isAsynchronous : CallAction \longrightarrow BOOLEAN$ determines whether a call action is asynchronous or not.

According to the UML metamodel, each action is identified by a name and a body instance of the metaclass `ActionExpression` which is left abstract and without semantics. As with operations, we give semantics to actions by means of ASM rules. We refine *action*(*trans*) in the *stateMachineExecution* as follows:

action(*trans*) ≡ **loop through** *act* ∈ *action*(*trans*)
case *act*
isSendAction : *send*(*obj*,*op*(*parList*))
isCallAction : *call*(*obj*,*op*(*parList*))
isCreateAction : *create*(*obj*,*parList*)
isTerminateAction : *destroy*(*self*)
isDestroyAction : *destroy*(*obj*)
isReturnAction : *return*(*obj*,*parList*)
isAssignmentAction : *act*
endcase

where *act* is an ASM function update if the action is an assignment, or has the form *op*(*obj*,*parList*) (i.e. *obj*.*op*(*parList*) in object oriented notation) otherwise.

The special macros formalizing the action execution are given below. Due to lack of space, we present here only those for call and send actions. They have the effect of updating the *IntEvQueue* of the target agent, since an action of the sender will result into an event for the target agent.

send(*obj*,*sig*) ≡ *insert*(*sig*,*IntEvQueue*(*obj*))

call(*obj*,*op*) ≡ **if** *isAgent*(*obj*)
then *insert*(*op*,*IntEvQueue*(*obj*))
else *op*(*obj*)

[6] The metaclass `UninterpretedAction` will be represented by its specialized classes, e.g. in our example we have the set *AssignmentAction*.

3.6 Modeling Synchronous Actions

The model presented here only deals with asynchronous call events. ASM sub-machine semantics does not allow to interrupt a sub-machine execution. This feature is required to model the blocking effect when synchronous calls are invoked: the caller sub-machine needs to suspend until the control is returned by the callee machine. This limitation can be overcome introducing in turbo ASMs suitable constructs for synchronous communication. However, in this section we propose a way for dealing with synchronous call events.

When an agent is suspended as a result of a synchronous call action *act*, we put it in a *blocked* mode on *act*, otherwise it is in mode *active*. Since an agent cannot make any move when waiting for the control to come back, we guard the two main rules (**Transition Selection** and **Generate Completion Event**), together with all the macros, with the condition **if** *mode*(*Self*) = *active* **then** ...

During the run to completion step an agent performing a synchronous call action is blocked as effect of the redefined *call* macro:

```
call(obj,op,trans) ≡ if isAgent(obj)
                     then insert(op,IntEvQueue(obj))
                          if not isAsynchronous(op)
                          then mode := blocked(op,trans)
                     else op(obj)
```

In order to keep memory of the transition on which the agent blocks, we add the parameter *trans* to the macro signature.

The agent is unblocked when the monitored function *getCtrl* is updated by the target agent as soon as it performs the corresponding return action.

```
Rule Unblock StateMachineExecution ≡
if mode = blocked(op,trans) & getCtrl(op)
then seq mode := active
         updateAction(op,trans)
         enterState(FromS,target(trans))
         case target(trans)
              SequentialState: enterInitialState(target(trans))
              ConcurrentState: startConcurrComput(target(trans))
              HistoryState: restoreConfig(target(trans))
         endcase
```

where *updateAction* holds the sequence of actions labeling the transition starting from, but excluding, *op*.

4 Conclusion and Related Work

We have presented a rigorous schema to map UML metamodel elements into ASMs. Given a UML model consisting of class, object and state diagrams, our mapping provides the vocabulary and the rules of a multi-agent ASM formalizing the given model. In such a way, we integrate UML static and dynamic views,

and propose a compositional semantics for UML state machines through a formalization of the interaction mechanism among different machines via actions and events.

This work comes as part of a formalization effort necessary to build a simulator for UML models. The toolkit framework is presented in [8].

An approach to map UML metamodel elements into ASMs has been presented in [9]. However, this work only takes into account model elements from a static view, neglecting any relation to their actual, dynamic semantics. Our work comes from a different perspective. In fact, although in [4] we provide a formal semantics for UML state machines, this proved to be not enough when one wants to integrate UML structural and behavioral diagrams, and formalize the state machines interaction mechanism.

References

[1] *OMG Unified Modeling Languages Specification, version 1.4*, 2001. 243

[2] *UML 1.4 Notation*, 2001. (Published as part of [1]). 231

[3] *UML 1.4 Semantics*, 2001. (Published as part of [1]). 231, 232, 237, 238

[4] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML State Machines: making semantic variation points and ambiguities explicit. In *Proc. Semantic Foundations of Engineering Design Languages (SFEDL02) - ETAPS 2002*. 229, 230, 233, 236, 237, 238, 239, 243

[5] E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In Y. Gurevich et al., editor, *Abstract State Machines. Theory and Applications*, volume 1912 of *LNCS 1912*, pages 223–241. Springer, 2000. 233, 236, 237, 238, 239

[6] E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Gurevich Festschrift)*, number 1862 in LNCS, pages 41–60, 2000. 240

[7] A. Cavarra and E. Riccobene. Simulating UML Statecharts. In R. Moreno-Diaz and A. Quesada-Arencibia, editors, *EUROCAST 2001*, pages 224–227. February 2001. 229

[8] A. Cavarra, E. Riccobene, and P. Scandurra. A Framework to Simulate UML Models. Submitted for publication. 243

[9] I. Ober. More meaningful UML Models. In *TOOLS - 37 Pacific 2000*. IEEE Computer Society Press, 2000. 243

[10] Poseidon UML Tool. `http://www.gentleware.com`. 233

[11] The precise UML group. `http://www.cs.york.ac.uk/puml/`. 230

The Hidden Computation Steps of Turbo Abstract State Machines

Nicu G. Fruja and Robert F. Stärk

Computer Science Department, ETH Zürich
CH–8092 Zürich, Switzerland
{fruja,staerk}@inf.ethz.ch

Abstract. Turbo Abstract State Machines are ASMs with parallel and sequential composition and possibly recursive submachine calls. Turbo ASMs are viewed as black-boxes that can combine arbitrary many steps of one or more submachines into one big step. The intermediate steps of a turbo ASM are not observable from outside. It is not even clear what exactly the intermediate steps are, because the semantics of turbo ASMs is usually defined inductively along the call graph of the ASM and the structure of the rule bodies. The most important application of turbo ASMs are recursive algorithms. Such algorithms can directly be simulated on turbo ASMs without transforming them into multi-agent (distributed) ASMs. In this article we analyze the hidden intermediate steps of turbo ASMs and characterize them using PAR/SEQ trees. We also address the problem of the reserve in the presence of recursion and sequential composition. Turbo ASMs with return values are obtained by syntactic sugar.

1 Introduction

Börger and Schmid view in [7] the execution of a sequential rule of an ASM or a recursive submachine call as one atomic step that produces exactly one set of parallel updates. Their black-box view of subcomputations makes it possible to combine arbitrary many computation steps of a submachine into one big step. The intermediate states of a submachine call are hidden and inaccessible from outside. Since recursive ASMs allow arbitrary speed-ups in the black-box view, we call them turbo ASMs in this article.

The black-box view of sequential composition and recursive submachine allows a direct simulation of recursive algorithms. Gurevich's ASM thesis, however, is violated, since for example a turbo ASM for the MergeSort algorithm can sort a list in *one* step. Hence the obvious question is: what are the hidden intermediate steps between two main steps $\mathfrak{A}$ and $\mathfrak{B}$ of a turbo ASM?

$$\mathfrak{A} \to \underbrace{\sigma_0 \to \sigma_1 \to \ \cdots \ \to \sigma_n}_{?} \to \mathfrak{B}$$

The first idea is to add a stack to ASMs as it is done in implementations of imperative programming languages. This works well as long as recursive calls

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 244–262, 2003.
© Springer-Verlag Berlin Heidelberg 2003

occur sequentially. If many recursive calls are done in parallel, the structure of the stack becomes more complicated. It is no longer a linear list, but a PAR/SEQ tree with nodes labeled by PAR (parallel composition) and SEQ (sequential composition). The main technical contribution of this article is the analysis of PAR/SEQ trees.

The local states in a PAR/SEQ tree can all be different. Also the environments that assign values to the free variables of transition rules can be different in different nodes. Hence, the approach of Bolognesi and Börger in [5] for the intermediate states of *Abstract State Processes* (ASPs) cannot be applied. Abstract State Processes extend ASM programs by process-algebraic behavior expressions for non-deterministic and concurrent processes. The configurations of an ASP are pairs consisting of an algebraic structure and a program. In a single computation step the ASP produces a set of updates and a new program which is called *residual* program. The update set is then applied to the structure and the residual program is executed in the new state. The intermediate steps of a sequential execution or a submachine call (process instantiation) can be traced and observed. The white-box operator for the sequential composition, however, is quite different from the black-box operator **seq** that we consider here. If two sequential processes run in parallel, then in the white-box view they can interfere, whereas in the black-box view they run completely independent. In the white-box view, the processes do not have a local state. After each step, the local changes become visible in the global state.

The plan of this article is as follows. In Sect. 2 we review the MergeSort example as a motivation for studying turbo ASMs. We then summarize in Sect. 3 syntax and semantics of turbo ASMs. In Sect. 4 we consider recursive ASMs with arbitrary nesting of **par**, **seq** and **forall**. We prove the equivalence of the inductive big-step semantics of turbo ASMs with the small-step semantics for PAR/SEQ trees. In Sect. 5, we address the problem of the reserve in the context of recursion and sequential composition. We show how the inductive definition of the semantics of turbo ASMs can easily be extended to turbo ASMs with a reserve by simply adding additional constraints. Once the reserve is available, turbo ASMs with return values are obtained simply by syntactic sugar. Finally, in Sect. 6 we compare turbo ASMs with other approaches to recursion for ASMs.

2 The MergeSort Example

Moschovakis argues in [13] that algorithms are *recursive equations* while abstract machine models are just *implementations*, a special kind of algorithms. This contradicts in some sense Gurevich's ASM thesis [3, 11] that each algorithm can be simulated on its natural level of abstraction step-by-step by an ASM. As an example for the insufficiency of machine models, Moschovakis, mentions the MergeSort algorithm.

Consider the problem of sorting a finite list $x = [x_1, \ldots, x_n]$ with respect to a linear ordering $\leq$. The MergeSort algorithm can succinctly be defined by the

recursive equation

$$\texttt{msort}(x) = \begin{cases} x, & \text{if } |x| \leq 1; \\ \texttt{merge}(\texttt{msort}(\texttt{l}(x)), \texttt{msort}(\texttt{r}(x))), & \text{otherwise.} \end{cases} \tag{1}$$

where $|x|$ denotes the length of the list x and $\texttt{l}(x)$, $\texttt{r}(x)$ are the first and second halves of the list x. The function $\texttt{merge}(x, y)$ is also defined recursively by the equation

$$\texttt{merge}(x, y) = \begin{cases} y, & \text{if } |x| = 0; \\ x, & \text{else, if } |y| = 0; \\ \texttt{hd}(x) \cdot \texttt{merge}(\texttt{tl}(x), y), & \text{else, if } \texttt{hd}(x) \leq \texttt{hd}(y); \\ \texttt{hd}(y) \cdot \texttt{merge}(x, \texttt{tl}(y)), & \text{otherwise.} \end{cases} \tag{2}$$

Here $a \cdot x$ denotes the result of prepending the element a to the list x (the `cons` of Lisp); $\texttt{hd}(x)$ is the first element of the list x (the `car` of Lisp) and $\texttt{tl}(x)$ is the rest of the list (the `cdr` of Lisp).

Moschovakis asks the question: 'If algorithms are machines, then which machine is the MergeSort?' — Gurevich's answer to the question in [4, 12] is that recursion is implicitly distributed (multi-agent) computation. To model the recursive MergeSort algorithm one has to translate the equations (1) and (2) into a distributed ASM. The algorithm is then given by the (partially ordered) runs of the distributed ASM.

Most people, however, do not think on distributed computation when they see equations (1) and (2). Computer Science students, for example, learn that the equations have a *declarative* (denotational) and a *procedural* (operational) semantics. The declarative semantics is given by the least fixed-point of a continuous functional associated to the equations. The operational semantics is obtained by viewing the equations as term rewrite rules and applying certain reduction strategies (call-by-value, call-by-name, leftmost-innermost, parallel-innermost, etc.). Moreover, in courses about algorithms, the data to be sorted are not elements of an immutable list but are stored in a dynamic array and are sorted in place.

We show now that the declarative as well as the operational semantics of the equations (1) and (2) can be modeled by ASMs in a natural way without using the notion of distributed ASMs. The ASMs in Fig. 1 and Fig. 2 are alternative answers to Moschovakis' question. We do not claim that they solve the problem, but they show another possible approach to recursive ASMs not mentioned in [4]. Moreover, the turbo ASMs in Fig. 2 are used as examples later.

The idea of the ASM for the declarative semantics in Fig. 1 is that the finite stages of the least fixed-point computation are exactly the states of the ASM. The functions $\texttt{msort}(x)$ and $\texttt{merge}(x, y)$ are dynamic functions which are initialized with the value *undef* for all lists x, y. The ASM is monotonic in the sense that a location that is defined ($\neq$ *undef*) cannot change its contents in later steps of the run. The ASM is highly parallel. In the first step it computes $\texttt{msort}(x)$ for all lists x of length 1. In the second step $\texttt{merge}(x, y)$ is computed for all lists x, y of length 1. In the third step, $\texttt{msort}(x)$ is computed for all lists

```
DeclarativeMergeSort ≙
  forall x ∈ List do
    if |x| ≤ 1 then
      msort(x) := x
    else
      msort(x) := merge(msort(l(x)), msort(r(x)))

DeclarativeMerge ≙
  forall x, y ∈ List do
    if |x| = 0 then
      merge(x, y) := y
    else if |y| = 0 then
      merge(x, y) := x
    else if hd(x) ≤ hd(y) then
      merge(x, y) := hd(x) · merge(tl(x), y)
    else
      merge(x, y) := hd(y) · merge(x, tl(y))
```

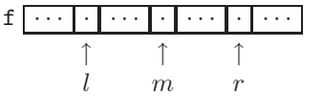
Fig. 1. The declarative semantics of MergeSort as a parallel ASM

```
Msort(l, r) ≙
  if l < r then
    let m = ⌊(l + r)/2⌋ in
      (Msort(l, m) par Msort(m + 1, r)) seq Merge(l, m, r)

Merge(l, m, r) ≙
  (forall i with l ≤ i ≤ r do g(i) := f(i)) seq MergeCopy(l, m, m + 1, r, l)

MergeCopy(i, m, j, r, k) ≙
  if k ≤ r then
    if (i ≤ m ∧ j ≤ r ∧ g(i) ≤ g(j)) ∨ (r < j) then
      f(k) := g(i) par MergeCopy(i + 1, m, j, r, k + 1)
    else
      f(k) := g(j) par MergeCopy(i, m, j + 1, r, k + 1)
```

Fig. 2. An implementation of MergeSort as a turbo ASM

of length 2, etc. The ASM runs forever. Hence, the ASM in Fig. 1 is not an implementation ASM but a pure specification ASM. One step in the run of the ASM corresponds to one step in the least fixed-point computation. (Note that the ASM thesis holds even in this rather abstract case.)

In Fig. 2 a turbo ASM is used for an implementation of the MergeSort algorithm. The ASM uses a unary dynamic function `f` that represents an array. The parameters l and r in $\textsc{Msort}(l, r)$ are the left and the right bound of the subarray $\texttt{f}(l\,..\,r)$ that has to be sorted.



The middle point m between l and r is computed and the subarrays $\texttt{f}(l\,..\,m)$ and $\texttt{f}(m+1\,..\,r)$ are recursively sorted in parallel by the calls $\textsc{Msort}(l, m)$ and $\textsc{Msort}(m+1, r)$. After that, the two sorted halves are merged together using $\textsc{Merge}(l, m, r)$. Thereby a second unary function `g` is used in which a copy of `f` is stored before the merging starts. $\textsc{MergeCopy}(i, m, j, r, k)$ merges the subarrays $\texttt{g}(i\,..\,m)$ and $\texttt{g}(j\,..\,r)$ into $\texttt{f}(k\,..\,r)$. The parameter i runs over the first half of `g` from l to m; the parameter j runs over the second half of `g` from $m+1$ to r and the parameter k runs over the whole array `f` from l to r.

Note that in the specification ASM in Fig. 1 the symbols `msort` and `merge` are names of dynamic functions, whereas in the implementation ASM in Fig. 2 the symbols Msort and Merge are names of (recursive) transition rules. In the specification ASM in Fig. 1, list operations are static functions. In the implementation ASM in Fig. 2, lists are represented by unary dynamic functions.

In the black-box view of turbo ASMs (as defined in [7]) the $\textsc{Msort}(l, r)$ in Fig. 2 sorts the subarray $\texttt{f}(l\,..\,r)$ in one step. This fact is rather surprising and the question arises what the intermediate steps of turbo ASMs exactly are. We try to answer this question in this article.

3 The Semantics of Recursive ASMs

Given a vocabulary Σ for predicate logic, we denote the terms of Σ by s, t and the first-order formulas of Σ by φ, ψ. The function names in Σ are declared either as static or dynamic. A turbo ASM consists of a finite set of rule declarations. A rule declaration for a rule name r is an expression $r(x) \triangleq P$, where P is a transition rule in which there are no free occurrences of variables except of x. Rule declarations can be recursive, i.e., the rule name r may appear in the body P like, for example, the rule name Msort in Fig. 2. The transition rules P, Q are syntactic expressions generated as follows (the function arguments can be read as vectors):

1. *Skip Rule:* **skip**
   Meaning: Do nothing.
2. *Update Rule:* $f(s) := t$
   Syntactic condition: f is a dynamic function name of Σ
   Meaning: In the next state, the value of f at s is updated to t.
3. *Block Rule:* P **par** Q
   Meaning: P and Q are executed in parallel.

4. *Conditional Rule:* **if** φ **then** P **else** Q
   Meaning: If φ is true, then execute P, otherwise execute Q.
5. *Let Rule:* **let** $x = t$ **in** P
   Meaning: Assign the value of t to x and execute P.
6. *Forall Rule:* **forall** x **with** φ **do** P
   Meaning: Execute P in parallel for each x satisfying φ.
7. *Choose Rule:* **choose** x **with** φ **do** P
   Meaning: Choose an x satisfying φ and execute P.
8. *Sequence Rule:* P **seq** Q
   Meaning: P and Q are executed sequentially, first P and then Q.
9. *Call Rule:* $r(t)$
   Meaning: Call transition rule r with parameters t.

The variables x in **let**, **forall** and **choose** are logical variables (also called *read only* variables). Its values cannot be updated by a transition rule. The values of the logical variables are not stored in the state but in a finite environment (also called *variable assignment*). The scope of x in **let** is the rule P (but not the term t), whereas the scope of x in **forall** or **choose** is φ and P.

The possible states of an ASM are the first-order structures for the vocabulary Σ. The semantics of transition rules is inductively defined in Table 1 by a predicate $\mathit{yields}(P, \mathfrak{A}, \zeta, U)$ with the meaning 'the transition rule P yields in state $\mathfrak{A}$ under the variables assignment ζ the update set U.' In the presence of the **choose** rule, a transition rule can yield several different update sets.

The following notations are used in Table 1: The value of a term t in a structure $\mathfrak{A}$ under a variable assignment ζ is denoted by $[\![t]\!]^{\mathfrak{A}}_{\zeta}$; the truth value of a formula φ by $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta}$. The range of a formula φ with distinguished variable x in a state $\mathfrak{A}$ and environment ζ is the set of all elements of $\mathfrak{A}$ that make the formula true:

$$\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta[x \mapsto a]} = \mathit{true}\}$$

By $\zeta[x \mapsto a]$ we denote the environment that is like ζ except for the variable x to which it assigns the element a.

A location of a state $\mathfrak{A}$ is a pair $\langle f, a\rangle$, where f is a function name of Σ and a is a list of elements of $\mathfrak{A}$. An update for $\mathfrak{A}$ is a pair $\langle l, v\rangle$, where l is a location and v (the new content of the location) is an element of $\mathfrak{A}$. The meaning of the update is that the function f is updated to the value v at the argument a.

An update set U is a set of updates. It is consistent, if it does not contain two updates for the same location with different values. If an update set U is consistent, then its updates can be applied to a state $\mathfrak{A}$. The result is a new state $\mathfrak{A} + U$ where the locations which are not updated in U keep their old contents.

The composition $U \oplus V$ of two update sets U and V is the set of updates obtained from U by adding the updates of V and overwriting updates in U which are redefined in V. If U and V are consistent, then $U \oplus V$ is consistent, too. The composition is defined such that the following equation is true for any state $\mathfrak{A}$:

$$\mathfrak{A} + (U \oplus V) = (\mathfrak{A} + U) + V$$

Table 1. Inductive definition of the semantics of turbo ASM rules

$\dfrac{}{\mathit{yields}(\mathbf{skip}, \mathfrak{A}, \zeta, \emptyset)}$	
$\dfrac{}{\mathit{yields}(f(s) := t, \mathfrak{A}, \zeta, \{\langle l, v\rangle\})}$	where $l = \langle f, a\rangle$, $a = [\![s]\!]^{\mathfrak{A}}_{\zeta}$, $v = [\![t]\!]^{\mathfrak{A}}_{\zeta}$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathit{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathit{yields}(P \ \mathbf{par}\ Q, \mathfrak{A}, \zeta, U \cup V)}$	
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta, U)}{\mathit{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta, U)}$	if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true}$
$\dfrac{\mathit{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathit{yields}(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta, V)}$	if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{false}$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathit{yields}(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, \zeta, U)}$	where $a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\mathit{yields}(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)}$	where $I = \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathit{yields}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta, U)}$	if $a \in \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$
$\dfrac{}{\mathit{yields}(\mathbf{choose}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta, \emptyset)}$	if $\mathit{range}(x, \varphi, \mathfrak{A}, \zeta) = \emptyset$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathit{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\mathit{yields}(P \ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U \oplus V)}$	if U is consistent
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta, U)}{\mathit{yields}(P \ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U)}$	if U is inconsistent
$\dfrac{\mathit{yields}(P\frac{t}{x}, \mathfrak{A}, \zeta, U)}{\mathit{yields}(r(t), \mathfrak{A}, \zeta, U)}$	where $r(x) \triangleq P$ is a rule declaration

The equation says that applying the update set $U \oplus V$ to state $\mathfrak{A}$ is the same as first applying U and then V.

How can it be that MSORT(l, r) in Fig. 2 sorts the subarray $\mathtt{f}(l\,..\,r)$ in one step? — We have to look at the rules for the sequential composition and at the rules for calls in Table 1. A sequential composition P **seq** Q is evaluated in state $\mathfrak{A}$ as follows. First, the rule P is evaluated in state $\mathfrak{A}$ and yields an update set U. If U is consistent, it is applied to $\mathfrak{A}$. The rule Q is then evaluated in the new state $\mathfrak{A} + U$ and yields an update set V. The result of the execution of P **seq** Q is the combined update set $U \oplus V$. After the completion of P **seq** Q the intermediate state $\mathfrak{A} + U$ is discarded. It is not visible any more.

A call $r(t)$ of a transition rule r with declaration $r(x) \triangleq P$ is evaluated by substituting the parameter t for x in the body P. The resulting transition rule $P\frac{t}{x}$ is evaluated in the same state. If it is defined and yields an update set,

then the update set is the result of the call. A call to a recursive rule that uses sequential compositions can generate an arbitrary number of intermediate states.

The substitution (call-by-name) semantics of rule calls has the advantage that it can easily be generalized to rule schemes that contain variables for transition rules which can be instantiated at run-time. Rule schemes have been used, for example, in [16] for the uniform specification of different versions of the Java Virtual Machine.

4 The Hidden Computation Steps of Turbo ASMs

The idea is that between two main states $\mathfrak{A}$ and $\mathfrak{B}$ of a turbo ASM there is a finite sequence of (hidden) intermediate states:

$$\mathfrak{A} \to \sigma_0 \to \sigma_1 \to \ \cdots \ \to \sigma_n \to \mathfrak{B}$$

The special cases of parallel ASMs (without **seq** composition) and sequential ASMs (without **par** and **forall** compositions) are described in the full version of this paper (available on the web). In the general case, several sequential ASMs can run in parallel as shown in the following example:

$$P \ \mathbf{seq}\ ((Q_1 \ \mathbf{seq}\ Q_2) \ \mathbf{par}\ (R_1 \ \mathbf{seq}\ R_2))$$

According to the semantics of rules in Table 1, the update set of this transition rule in a state $\mathfrak{A}$ is $U \oplus ((V_1 \oplus V_2) \cup (W_1 \oplus W_2))$, if

1. P yields U in $\mathfrak{A}$ and U is consistent,
2. Q_1 yields V_1 in $\mathfrak{A} + U$ and V_1 is consistent,
3. Q_2 yields V_2 in $(\mathfrak{A} + U) + V_1$,
4. R_1 yields W_1 in $\mathfrak{A} + U$ and W_1 is consistent,
5. R_2 yields W_2 in $(\mathfrak{A} + U) + W_1$.

The example shows that, in general, the update set that has to be added to the initial state $\mathfrak{A}$ in order to obtain the current state and the update set to which the result of a transition rule has to be added are not the same. For example, the transition rule Q_2 is evaluated in state $\mathfrak{A} + (U \oplus V_1)$, but the result set V_2 has to be added to V_1 and not to $U \oplus V_1$.

Therefore, the intermediate states between two main states of a turbo ASM are trees (a kind of generalized stacks). The leaves of the trees are either update sets or frames containing rule calls or sequential compositions. A *frame* is a triple $\langle P, \mathfrak{A}, \zeta \rangle$ consisting of a transition rule P, a local state $\mathfrak{A}$, and an environment ζ. The internal nodes of the trees are labeled with PAR or SEQ. The trees are called PAR/SEQ trees. The nodes can be pictured as follows:

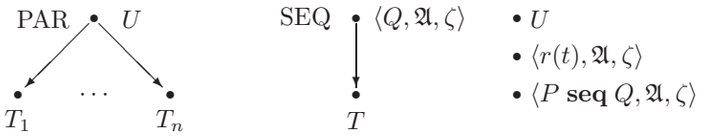


Table 2. Partial evaluation of ASM rules

$$
\begin{aligned}
&ev(\mathbf{skip}, \mathfrak{A}, \zeta) = \emptyset \\
&ev(f(s) := t, \mathfrak{A}, \zeta) = \{\langle\langle f, a\rangle, b\rangle\} \quad \text{where } a = [\![s]\!]^{\mathfrak{A}}_{\zeta} \text{ and } b = [\![t]\!]^{\mathfrak{A}}_{\zeta} \\
&ev(P \ \mathbf{par}\ Q, \mathfrak{A}, \zeta) = ev(P, \mathfrak{A}, \zeta) \cup ev(Q, \mathfrak{A}, \zeta) \\
&ev(\mathbf{if}\ \varphi\ \mathbf{then}\ P\ \mathbf{else}\ Q, \mathfrak{A}, \zeta) = \begin{cases} ev(P, \mathfrak{A}, \zeta), & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{true}; \\ ev(Q, \mathfrak{A}, \zeta), & \text{otherwise.} \end{cases} \\
&ev(\mathbf{let}\ x = t\ \mathbf{in}\ P, \mathfrak{A}, \zeta) = ev(P, \mathfrak{A}, \zeta[x \mapsto a]) \quad \text{where } a = [\![t]\!]^{\mathfrak{A}}_{\zeta} \\
&ev(\mathbf{forall}\ x\ \mathbf{with}\ \varphi\ \mathbf{do}\ P, \mathfrak{A}, \zeta) = \textstyle\bigcup_{a \in range(x, \varphi, \mathfrak{A}, \zeta)} ev(P, \mathfrak{A}, \zeta[x \mapsto a]) \\
&ev(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta) = \{\langle P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta\rangle\} \\
&ev(r(t), \mathfrak{A}, \zeta) = \{\langle r(t), \mathfrak{A}, \zeta\rangle\}
\end{aligned}
$$

Each PAR node has an update set. The children of the PAR node are executed in parallel. The union of their result sets is added to the update set stored in the PAR node. A PAR node with update set U and children $T_1, \ldots, T_n$ is written as $\mathrm{PAR}(U, [T_1, \ldots, T_n])$. If each subtree T_i yields an update set V_i, then the PAR node yields the update set $U \oplus (V_1 \cup \ldots \cup V_n)$.

A SEQ node has a frame $\langle Q, \mathfrak{A}, \zeta\rangle$ and a single child T. It is written as $\mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)$. If the child T yields a consistent update set U, then Q is executed in the state $\mathfrak{A} + U$. The resulting update set is then added to U. Formally, PAR/SEQ trees are defined by the following grammar:

$$T ::= U \mid \langle r(t), \mathfrak{A}, \zeta\rangle \mid \langle P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta\rangle \mid \mathrm{PAR}(U, [T_1, \ldots, T_n]) \mid \mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)$$

To describe the transitions between PAR/SEQ trees we introduce partial evaluation of ASM rules. The result of the partial evaluation of an ASM rule P in a state $\mathfrak{A}$ and an environment ζ is denoted by $ev(P, \mathfrak{A}, \zeta)$ and defined in Table 2. The function ev is recursively defined on the structure of ASM rules. It leaves rule calls and sequential compositions unevaluated.

Since in our notation $ev(P, \mathfrak{A}, \zeta)$ denotes a set and the PAR trees are defined using a list of children (given by a partial evaluation as we will see in the rewrite rules below), we introduce $\overline{ev}(P, \mathfrak{A}, \zeta)$ to represent the list of all elements from the corresponding $ev(P, \mathfrak{A}, \zeta)$ (updates are combined into a single set), i.e.

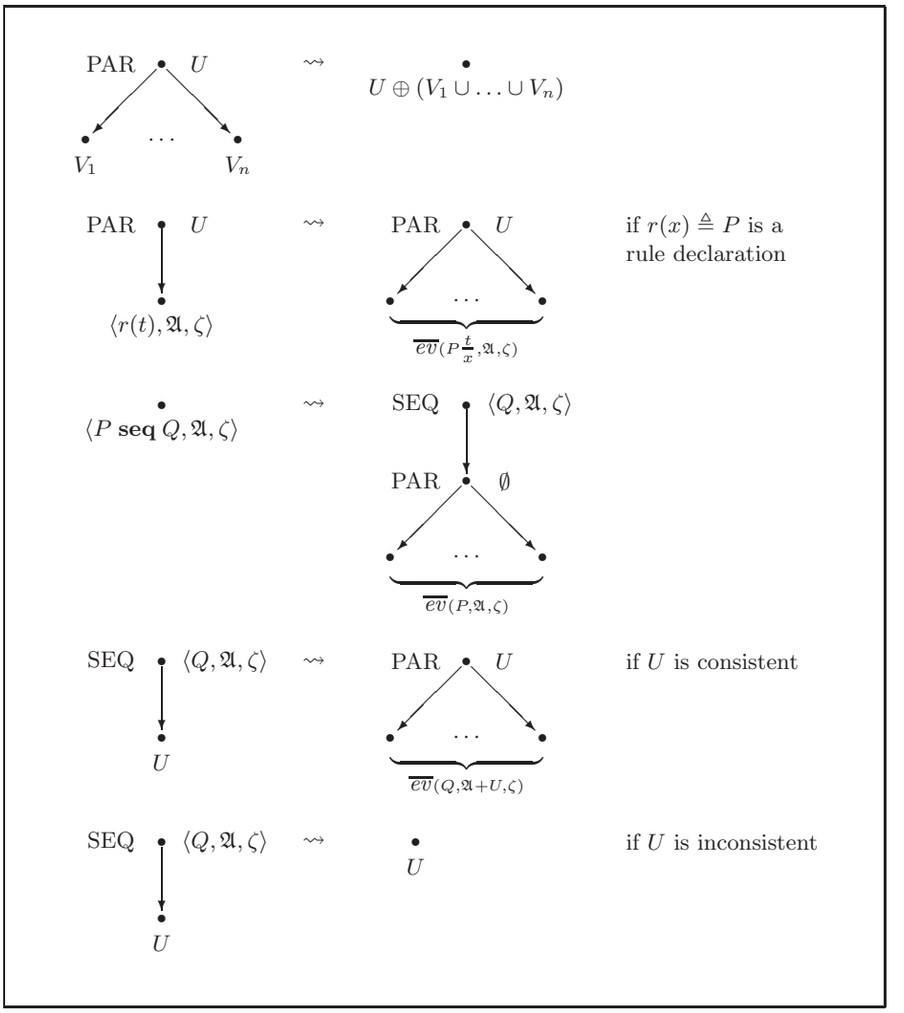
$$\overline{ev}(P, \mathfrak{A}, \zeta) = [U, F_1, \ldots, F_k], \text{ if } ev(P, \mathfrak{A}, \zeta) = U \cup \{F_1, \ldots, F_k\},$$

where U is an update set and $F_1, \ldots, F_k$ are frames.

The first intermediate state σ_0 is the PAR tree consisting of an empty update set and the children given by the partial evaluation of the main rule of the ASM in the state $\mathfrak{A}$:

$$\sigma_0 = \mathrm{PAR}(\emptyset, \overline{ev}(P_0, \mathfrak{A}, \zeta_0))$$

Table 3. Graphical representation for the rewrite rules



The last intermediate state σ_n is a tree consisting of a single node hosting an update set U. The next main state $\mathfrak{B}$ is obtained from $\mathfrak{A}$ by applying the updates from U, i.e.

$$\mathfrak{B} = \mathfrak{A} + U$$

Since the intermediate states are given as trees, the transition between trees is described by means of *rewrite rules* for trees, i.e. given an intermediate state σ_i,

the next intermediate state σ_{i+1} is determined by applying simultaneously the rules from Table 3 to the tree that represents σ_i.

Note that, because of the rewrite rules, all the intermediate states except the last are PAR trees. The SEQ trees may occur only as subtrees of these intermediate states. The subtrees of an intermediate PAR tree are transformed simultaneously as follows:

1. If the children of a PAR node with update set U are update sets $V_1, \ldots, V_n$, then the PAR node is replaced by the update set obtained from U by adding the union $V_1 \cup \ldots \cup V_n$.
2. If a leaf is a rule call, then it is replaced by the partial evaluation of the rule obtained by substituting the formal parameters in the corresponding rule body.
3. If a leaf is a sequential composition $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta\rangle$, then it is replaced by the SEQ tree with the frame $\langle Q, \mathfrak{A}, \zeta\rangle$ and the child given by the PAR tree with an empty update set and children given by the partial evaluation of P.
4. If a subtree is a SEQ tree with the frame $\langle Q, \mathfrak{A}, \zeta\rangle$ and its child is a consistent update set U, then it is replaced by the PAR tree with the update set U and the children given by the partial evaluation of Q in the state $\mathfrak{A} + U$ under ζ.
5. If a subtree is a SEQ tree and its child is an inconsistent update set U, then it is replaced by the tree consisting of a single node given by U.

We say that a tree S is a successor of a tree T (written $T \rightsquigarrow S$) iff the tree S is obtained by applying the rewrite rules to T. Since we do not deal only with single trees, but also with lists of trees, we inductively define what it means that a tree T rewrites to a list of trees $S_1, \ldots, S_k$ (written $T \rightsquigarrow [S_1, \ldots, S_k]$):

1. If U is an update set, then $U \rightsquigarrow [U]$.
2. If $r(x) \triangleq P$ is a rule declaration, then $\langle r(t), \mathfrak{A}, \zeta\rangle \rightsquigarrow \overline{ev}(P\frac{t}{x}, \mathfrak{A}, \zeta)$.
3. $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta\rangle \rightsquigarrow [\mathrm{SEQ}(\mathfrak{A}, \mathrm{PAR}(\emptyset, \overline{ev}(P, \mathfrak{A}, \zeta)), Q, \zeta)]$.
4. If $V_1, \ldots, V_n$ are update sets, then $\mathrm{PAR}(U, [V_1, \ldots, V_n]) \rightsquigarrow [U \oplus (V_1 \cup \ldots \cup V_n)]$.
5. If $L_1, \ldots, L_n$ are lists of trees such that $T_i \rightsquigarrow L_i$ for all $i = 1, \ldots, n$ and not all of $T_1, \ldots, T_n$ are update sets, then

$$\mathrm{PAR}(U, [T_1, \ldots, T_n]) \rightsquigarrow [\mathrm{PAR}(U, L_1 \cdot \ldots \cdot L_n)]$$

   (where $L_1 \cdot \ldots \cdot L_n$ denotes the concatenation of the lists $L_1, \ldots, L_n$).
6. If S is not an update set and $S \rightsquigarrow [T]$, then

$$\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rightsquigarrow [\mathrm{SEQ}(\mathfrak{A}, T, Q, \zeta)].$$

7. If U is a consistent update set, then

$$\mathrm{SEQ}(\mathfrak{A}, U, Q, \zeta) \rightsquigarrow [\mathrm{PAR}(U, \overline{ev}(Q, \mathfrak{A} + U, \zeta))].$$

8. If U is an inconsistent update set, then $\mathrm{SEQ}(\mathfrak{A}, U, Q, \zeta) \rightsquigarrow [U]$.

According to the previous recursive definition, if T and S are trees such that $T \rightsquigarrow [S]$, then S is a successor of T. We denote by $\rightsquigarrow^*$ the transitive closure of $\rightsquigarrow$ and consider the natural extension of $\rightsquigarrow$ (and implicitly $\rightsquigarrow^*$) for lists of trees.

One direction of the main result can be proved by induction on the definition of 'yields': if a (general) turbo ASM can make one big step from $\mathfrak{A}$ to $\mathfrak{B}$ then, there exists a finite sequence of intermediate states as described above.

Lemma 1. *Let P be a rule of a general turbo ASM. If P yields U in $\mathfrak{A}$ under ζ, then there exist update sets $V_1, \ldots, V_k$ such that*

$$\overline{ev}(P, \mathfrak{A}, \zeta) \rightsquigarrow^* [V_1, \ldots, V_k] \quad \textit{and} \quad U = V_1 \cup \ldots \cup V_k.$$

Proof. By induction on the definition of $\mathit{yields}(P, \mathfrak{A}, \zeta, U)$ in Table 1. □

For the other direction we have to define what it means that a PAR/SEQ tree T yields an update set U (written $T \rhd U$). The relation $T \rhd U$ is inductively defined as follows :

1. $U \rhd U$.
2. If $\mathit{yields}(r(t), \mathfrak{A}, \zeta, U)$, then $\langle r(t), \mathfrak{A}, \zeta\rangle \rhd U$.
3. If $\mathit{yields}(P \textbf{ seq } Q, \mathfrak{A}, \zeta, U)$, then $\langle P \textbf{ seq } Q, \mathfrak{A}, \zeta\rangle \rhd U$.
4. If $T = \mathrm{PAR}(U, [T_1, \ldots, T_n])$ and $T_i \rhd V_i$ for all $1 \leq i \leq n$, then $T \rhd U \oplus (V_1 \cup \ldots \cup V_n)$.
5. If $S \rhd U$ with U consistent and $\mathit{yields}(Q, \mathfrak{A} + U, \zeta, V)$, then $\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rhd U \oplus V$.
6. If $S \rhd U$ with U inconsistent, then $\mathrm{SEQ}(\mathfrak{A}, S, Q, \zeta) \rhd U$.

The following result relates the partial evaluation of a rule with the set of updates it yields.

Lemma 2. *Let P be a rule of a general turbo ASM. If*

$$ev(P, \mathfrak{A}, \zeta) = U \cup \{\langle P_i, \mathfrak{A}, \eta_i\rangle \mid 1 \leq i \leq k\}$$

where U is an update set and $\mathit{yields}(P_i, \mathfrak{A}, \eta_i, V_i)$ for each $i = 1, \ldots, k$, then the following holds:

$$\mathit{yields}(P, \mathfrak{A}, \zeta, U \cup \bigcup_{1 \leq i \leq k} V_i)$$

Proof. By induction on the size of P using the definition of $ev()$ in Table 2. □

A tree that rewrites to a list of trees, yields an update set, that is the same with the union of the update sets the trees from the list yield:

Lemma 3. *If T and $S_1, \ldots, S_k$ are trees such that $T \rightsquigarrow [S_1, \ldots, S_k]$ and $S_i \rhd V_i$ for all $i \leq k$, then*

$$T \rhd \bigcup_{1 \leq i \leq k} V_i.$$

Proof. By induction on the definition of $T \rightsquigarrow [S_1, \ldots, S_k]$. We use Lemma 2 for the cases when T has one of the following forms:

1. $T = \langle r(t), \mathfrak{A}, \zeta \rangle$,
2. $T = \langle P \textbf{ seq } Q, \mathfrak{A}, \zeta \rangle$,
3. $T = \mathrm{SEQ}(\mathfrak{A}, U, Q, \zeta)$ with U consistent.

We need the result of the lemma only for the particular case:

T, S are intermediate states (PAR trees or update sets) such that $T \rightsquigarrow [S]$,

but since the proof is based on the inductive definition of $T \rightsquigarrow [S_1, \ldots, S_k]$, one has to prove the lemma for all the trees. □

Now, we are able to prove the other direction of the main result: if there exists a finite sequence of rewrite steps that ends into an update set U, then a turbo ASM can make one big step from $\mathfrak{A}$ to $\mathfrak{A} + U$.

Lemma 4. *Let P be a rule of a general turbo ASM. If there exists a finite sequence of successors from $\mathrm{PAR}(\emptyset, \overline{ev}(P, \mathfrak{A}, \zeta))$ to an update set U, then P yields U in $\mathfrak{A}$ under ζ.*

Proof. If $(\sigma_i)_{0 \le i \le n}$ is a sequence of trees such that

1. $\sigma_0 = \mathrm{PAR}(\emptyset, \overline{ev}(P, \mathfrak{A}, \zeta))$,
2. $\sigma_i \rightsquigarrow \sigma_{i+1}$ for each $i < n$,
3. $\sigma_n = U$,

then, using Lemma 3 for the case $k = 1$, it can be proved by induction on $n - i$ that $\sigma_i \rhd U$ for each $i \le n$. Then, the proof is done, since $\mathrm{PAR}(\emptyset, \overline{ev}(P, \mathfrak{A}, \zeta)) \rhd U$ implies $\mathit{yields}(P, \mathfrak{A}, \zeta, U)$ (Lemma 2). □

5 The Reserve and Return Values

Algorithms often need to increase their working space. ASMs therefore have by definition (Lipari Guide [9]) an infinite reserve which is part of the base set. New elements are allocated using the construct

import x **do** P

which means 'choose an element x from the reserve, delete it from the reserve and execute P'. The reserve of a state is represented using a special dynamic predicate (boolean valued function) `Reserve` such that the reserve elements $\mathrm{Res}(\mathfrak{A})$ of a state $\mathfrak{A}$ is the set of elements a of $\mathfrak{A}$ such that $\texttt{Reserve}(a) = \mathit{true}$ in $\mathfrak{A}$.

The special dynamic function `Reserve` is avoided in [2, 10]. In [10] the reserve of a state is defined to be the base set of the state minus the contents of locations and the elements (arguments) of defined ($\neq \mathit{undef}$) locations. In the presence of sequential composition, however, the use of a special dynamic function `Reserve` seems to be unavoidable.

Why? First, one has to ensure that, if new elements are imported in parallel, they are different. For example, after executing the rule

$$(\textbf{import } x \textbf{ do } f(x) := 0) \textbf{ par } (\textbf{import } x \textbf{ do } f(x) := 0)$$

Table 4. The semantics of turbo ASMs with a reserve

$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathit{yields}(\textbf{import } x \textbf{ do } P, \mathfrak{A}, \zeta, V)}$	if $a \in \mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ and $V = U \cup \{\langle\langle \texttt{Reserve}, a\rangle, \mathit{false}\rangle\}$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathit{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathit{yields}(P \textbf{ par } Q, \mathfrak{A}, \zeta, U \cup V)}$	if $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U) \cap \mathrm{El}(V) \subseteq \mathrm{ran}(\zeta)$
$\dfrac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U_a) \quad \text{for each } a \in I}{\mathit{yields}(\textbf{forall } x \textbf{ with } \varphi \textbf{ do } P, \mathfrak{A}, \zeta, \bigcup_{a \in I} U_a)}$	if $I = \mathit{range}(x, \varphi, \mathfrak{A}, \zeta)$ and for $a \neq b$ $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U_a) \cap \mathrm{El}(U_b) \subseteq \mathrm{ran}(\zeta)$

there exist two different elements a and b with $f(a) = 0$ and $f(b) = 0$. That different occurrences of **import** get different reserve elements can only be ensured from outside by requiring, for example, in a **par** rule that the reserve elements of the update sets of the two components are disjoint.

Now, if we execute a sequential rule P **seq** Q and we do not use the special predicate `Reserve` as it is done in [10], it can happen that an element which is not in the reserve in the initial state, is again in the reserve after executing P (for example, if P updates all locations that contain the element to *undef*). Hence, the element can be imported in Q. In the combined update set of P and Q, however, the element does no longer count as a reserve element, since it is not in the reserve of the initial state but only in the reserve of the intermediate state (which is hidden from outside). Therefore, if we execute two copies of P **seq** Q in parallel, it could be that both processes import the same element inside Q without violating the condition that in a union of update sets the reserve elements are disjoint.

For this reason we use the special dynamic function `Reserve` and say that a state $\mathfrak{A}$ satisfies the *reserve condition* with respect to an environment ζ, if the following two conditions hold for each element $a \in \mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$:

R1. The element a is not the contents of a location of $\mathfrak{A}$.
R2. If a is an element of a location l of $\mathfrak{A}$ which is not a location for `Reserve`, then the contents of l in $\mathfrak{A}$ is *undef*.

The new rules for **import** are listed in Table 4. Rather than using *actions* (equivalence classes of update sets modulo permutations of the reserve, see [10]) instead of update sets, we add additional constraints to the rules for **par** and **forall**. By $\mathrm{El}(U)$ we denote the set of elements that occur in the updates of U. The elements of a location $\langle f, \langle a_1, \ldots, a_n\rangle\rangle$ are the arguments $a_1, \ldots, a_n$. The elements of an update $\langle l, v\rangle$ are the elements of the location l and the value v.

When a new element a is imported in the rule for **import** we require that a is an element of the reserve of $\mathfrak{A}$ but does not occur in the range of the variable assignment ζ which contains possibly other new elements that are imported in the same step. The value of the variable x is redefined to a and the body P

of the **import** rule is executed in the new environment. The function `Reserve` is updated at the argument a to *false* (by the system and not directly by the ASM). The constraint for P **par** Q says that the reserve elements of the update set U for P are disjoint from the reserve elements of the update set V for Q except for reserve elements in the range of ζ. We have to exclude the range of ζ, since otherwise the execution of a rule like

$$\textbf{import } x \textbf{ do } (f(x) := 0 \textbf{ par } g(x) := 0)$$

would be impossible. In order that the constraints in Table 4 work correctly, we have to require that in the scope of bound variable the same variable is not used again as a bound variable (in a **let**, **forall**, **choose**, or **import**). Otherwise, it could happen that a reserve element that is imported and bound to a variable in the environment is hidden and then imported again. Example:

import x **do**
  $f(x) := 0$ **par**
  let $x = 1$ **in import** y **do** $f(y) := x$

In this example, the same reserve element could be used for x as well as for y, since the outermost x is hidden by the **let** and not visible in the environment, when y is imported.

Lemma 5 (Preservation of the Reserve Condition). *If a state* $\mathfrak{A}$ *satisfies the reserve condition wrt.* ζ *and* P *yields a consistent update set* U *in* $\mathfrak{A}$ *under* ζ*, then* $\mathfrak{A} + U$ *satisfies the reserve condition wrt.* ζ *and* $\mathrm{Res}(\mathfrak{A} + U) \setminus \mathrm{ran}(\zeta)$ *is contained in* $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{El}(U)$*.*

Proof. One first shows by induction on the definition of 'yields' that, if P yields U in $\mathfrak{A}$ under ζ and a is an element of U, then a is the content of a location in $\mathfrak{A}$ or a is in $\mathrm{ran}(\zeta)$ or the update $\langle\langle \texttt{Reserve}, a\rangle, \mathit{false}\rangle$ is in U. Thereby, one uses the fact that the value of a term t in $\mathfrak{A}$ under ζ is the content of a location of $\mathfrak{A}$ or an element of the range of ζ. □

If a structure $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, then a permutation of the set $\mathrm{Res}(\mathfrak{A})\setminus\mathrm{ran}(\zeta)$ can always be extended to an automorphism of the structure $\mathfrak{A}$ which is the identity on the non-reserve elements and the elements in the range of ζ. Hence, the following lemma can be used to rename the reserve elements using a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$.

Lemma 6 (Preservation of Isomorphisms). *If* α *is an isomorphism from* $\mathfrak{A}$ *to* $\mathfrak{B}$ *and* P *yields* U *in* $\mathfrak{A}$ *under* ζ*, then* P *yields* $\alpha(U)$ *in* $\mathfrak{B}$ *under* $\alpha \circ \zeta$*.*

We can now prove that the update set computed by a transition rule in a state is unique modulo permutations of the reserve (if no **choose** is used). This is the justification for the constraints in Table 4.

Lemma 7 (Independence of the Choice of Reserve Elements). *Let P be a rule of an ASM without* **choose**. *If $\mathfrak{A}$ satisfies the reserve condition wrt. ζ, the bound variables of P are not in the domain of ζ, P yields U in $\mathfrak{A}$ under ζ, and P yields U' in $\mathfrak{A}$ under ζ, then there exists a permutation α of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\alpha(U) = U'$.*

Proof. By induction on the definition of 'yields'. We consider the critical cases.

Case **import**: Assume that $a, a' \in \mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ and

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a], U)}{\mathit{yields}(\mathbf{import}\ x\ \mathbf{do}\ P, \mathfrak{A}, \zeta, U \cup \{\langle\langle \mathtt{Reserve}, a\rangle, \mathit{false}\rangle\})}$$

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a'], U')}{\mathit{yields}(\mathbf{import}\ x\ \mathbf{do}\ P, \mathfrak{A}, \zeta, U' \cup \{\langle\langle \mathtt{Reserve}, a'\rangle, \mathit{false}\rangle\})}$$

Let α be the permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ that transposes a and a'.

Since α is an automorphism of $\mathfrak{A}$ and $\alpha \circ (\zeta[x \mapsto a]) = \zeta[x \mapsto a']$, we can apply Lemma 6 and obtain $\mathit{yields}(P, \mathfrak{A}, \zeta[x \mapsto a'], \alpha(U))$.

Since $x \notin \mathrm{dom}(\zeta)$, we have $\mathrm{ran}(\zeta[x \mapsto a']) = \mathrm{ran}(\zeta) \cup \{a'\}$ and therefore $\mathfrak{A}$ satisfies the reserve condition wrt. $\zeta[x \mapsto a']$. By the induction hypothesis, there exists a permutation β of $\mathrm{Res}(\mathfrak{A}) \setminus (\mathrm{ran}(\zeta) \cup \{a'\})$ such that $\beta(\alpha(U)) = U'$. Since $\beta \circ \alpha$ is a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ and $\beta(\alpha(a)) = a'$, we are done.

Case **par**: Assume that

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathit{yields}(Q, \mathfrak{A}, \zeta, V)}{\mathit{yields}(P\ \mathbf{par}\ Q, \mathfrak{A}, \zeta, U \cup V)} \qquad \frac{\mathit{yields}(P, \mathfrak{A}, \zeta, U') \quad \mathit{yields}(Q, \mathfrak{A}, \zeta, V')}{\mathit{yields}(P\ \mathbf{par}\ Q, \mathfrak{A}, \zeta, U' \cup V')}$$

and $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U) \cap \mathrm{El}(V) \subseteq \mathrm{ran}(\zeta)$ and $\mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U') \cap \mathrm{El}(V') \subseteq \mathrm{ran}(\zeta)$.

By the induction hypothesis there exist permutations α and β of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\alpha(U) = U'$ and $\beta(V) = V'$.

Let γ be the restriction of $\alpha \cup \beta$ to $\mathrm{El}(U) \cup \mathrm{El}(V)$. The function γ is well-defined, since if $a \in \mathrm{El}(U) \cap \mathrm{El}(V) \cap \mathrm{Res}(\mathfrak{A})$, then $a \in \mathrm{ran}(\zeta)$ and $\alpha(a) = a = \beta(a)$.

The function γ is injective. Assume that $a \in \mathrm{El}(U)$, $b \in \mathrm{El}(V)$ and $\alpha(a) = \beta(b)$. Let $c := \alpha(a)$. Then $c \in \mathrm{Res}(\mathfrak{A}) \cap \mathrm{El}(U') \cap \mathrm{El}(V')$ and therefore $c \in \mathrm{ran}(\zeta)$ and we obtain $a = \alpha(a) = c = \beta(b) = b$.

Hence there exists a permutation δ of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\delta(U) = U'$, $\delta(V) = V'$ and $\delta(U \cup V) = U' \cup V'$.

Case **seq**: Assume that U and U' are consistent and

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta, U) \quad \mathit{yields}(Q, \mathfrak{A} + U, \zeta, V)}{\mathit{yields}(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U \oplus V)}$$

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta, U') \quad \mathit{yields}(Q, \mathfrak{A} + U', \zeta, V')}{\mathit{yields}(P\ \mathbf{seq}\ Q, \mathfrak{A}, \zeta, U' \oplus V')}$$

```
MSORT(x) ≜
  if |x| = 1 then return x
  else
    let y = MSORT(l(x)), z = MSORT(r(x)) in
      let r = MERGE(y, z) in return r

MERGE(x, y) ≜
  if |x| = 0 then return y
  else if |y| = 0 then return x
  else if hd(x) ≤ hd(y) then
    let z = MERGE(tl(x), y) in return hd(x) · z
  else
    let z = MERGE(x, tl(y)) in return hd(y) · z
```

Fig. 3. MergeSort as a turbo ASM with return values

By the induction hypothesis there exists a permutation α of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$ such that $\alpha(U) = U'$. Since α is an automorphism of $\mathfrak{A}$, it is an isomorphism of $\mathfrak{A}+U$ to $\mathfrak{A}+U'$. Since $\alpha \circ \zeta = \zeta$, by Lemma 6, it follows that $\mathit{yields}(Q, \mathfrak{A}+U', \zeta, \alpha(V))$.

By Lemma 5, it follows that $\mathfrak{A} + U'$ satisfies the reserve condition wrt. ζ and $\mathrm{Res}(\mathfrak{A} + U') \setminus \mathrm{ran}(\zeta)$ is contained in $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{El}(U')$.

By the induction hypothesis, there exists a permutation β of $\mathrm{Res}(\mathfrak{A}+U') \setminus \mathrm{ran}(\zeta)$ such that $\beta(\alpha(V)) = V'$.

Hence, $\beta \circ \alpha$ is a permutation of $\mathrm{Res}(\mathfrak{A}) \setminus \mathrm{ran}(\zeta)$. Since $\beta(U') = U'$, it follows that $(\beta \circ \alpha)(U \oplus V) = U' \oplus V'$. □

ASMs with return values are now obtained by syntactic sugar. We use the notation of Börger and Bolognesi in [6]. The expression **return** t is considered as an abbreviation for the update $\mathtt{result}(\mathit{this}) := t$, where `result` is special unary dynamic function that is used for returning values. The variable *this* is a special variable which is implicit in each rule declaration as argument 0. The value of *this* is the location into which the return value has to be stored. If we have a rule declaration $r(x) \triangleq P$, we write $s.r(t)$ to indicate that the variable *this* is bound to s and x is bound to t. We write

$$\textbf{let } x_1 = r_1(t_1), \ldots, x_n = r_n(t_n) \textbf{ in } Q$$

as a syntactic abbreviation for

$$\begin{aligned}&\textbf{import } y_1, \ldots, y_n \textbf{ do}\\ &\quad (y_1.r_1(t_1) \textbf{ par } \ldots \textbf{ par } y_n.r_n(t_n)) \textbf{ seq}\\ &\quad \textbf{let } x_1 = \mathtt{result}(y_1), \ldots, x_n = \mathtt{result}(y_n) \textbf{ in } Q\end{aligned}$$

First, new locations are created for storing the return values. Then the transition rules with return values are called in parallel and store the results in the new

Table 5. Inductive definition of the semantics of Xasm rule calls

$$\frac{}{\mathit{run}(P, \mathfrak{A}, \zeta, \mathfrak{A})}$$

$$\frac{\mathit{run}(P, \mathfrak{A}, \zeta, \mathfrak{B}) \quad \mathit{yields}(P, \mathfrak{B}, \zeta, U)}{\mathit{run}(P, \mathfrak{A}, \zeta, \mathfrak{B} + U)} \text{ if } U \text{ is consistent}$$

$$\frac{\mathit{yields}(P, \mathfrak{A}, \zeta, \emptyset)}{\mathit{final}(P, \mathfrak{A}, \zeta)}$$

$$\frac{\mathit{run}(P, \mathfrak{A}, \eta, \mathfrak{B}) \quad \mathit{final}(P, \mathfrak{B}, \eta)}{\mathit{yields}(r(t), \mathfrak{A}, \zeta, \mathfrak{B} - \mathfrak{A})} \quad \begin{array}{l}\text{where } r(x) \triangleq P \text{ is a rule declaration,} \\ a = [\![t]\!]^{\mathfrak{A}}_{\zeta} \text{ and } \eta = \zeta[x \mapsto a]\end{array}$$

locations. After that, the results of the executions are bound to the **let**-variables and are then used in the body Q.

We are now in the position to present another turbo ASM for MergeSort. The ASM in Fig. 3 uses return values and sorts an immutable list. The ASM is a direct translation of the equations (1) and (2) in Sect. 2.

6 Recursion in Other Implementations of ASMs

The inductive semantics for turbo ASMs in Table 1 describes the semantical basis for Schmid's AsmGofer system [14]. A slightly different approach to recursion is taken in Xasm [1]. In this section, we want to point out the essential differences. We focus just on recursion and omit other features of Xasm like the evaluation of terms with side effects (update sets) or the so-called external functions of Xasm. Unfortunately, it is not clear to us which kind of recursion is supported by AsmL [8].

For the definition of the semantics of recursive calls in Xasm we need in addition to the predicate $\mathit{yields}(P, \mathfrak{A}, \zeta, U)$ two new predicates $\mathit{run}(P, \mathfrak{A}, \zeta, \mathfrak{B})$ and $\mathit{final}(P, \mathfrak{A}, \zeta)$. The predicate $\mathit{run}(P, \mathfrak{A}, \zeta, \mathfrak{B})$ means that there exists a finite run of P from $\mathfrak{A}$ into state $\mathfrak{B}$ where the free variables of P are defined in the environment ζ. The predicate $\mathit{final}(P, \mathfrak{A}, \zeta)$ means that $\mathfrak{A}$ is a final state for P under ζ. The predicates are defined by *simultaneous* induction. The clauses for *yields* are the same as in Table 1 except for rule calls. The new clauses are listed in Table 5.

How does a (recursive) rule call $r(t)$ in Xasm work? Assume that $r(x) \triangleq P$ is a rule declaration. First the argument t is evaluated in the current state $\mathfrak{A}$ and the current environment ζ. The value of t is assigned to the variable x in a new environment η. Then a new run is started for the body P in state $\mathfrak{A}$ with the environment η. If after finitely many steps, the run terminates in a final state $\mathfrak{B}$ (where P yields an empty update set), then the rule call $r(t)$ yields the difference

between the final state $\mathfrak{B}$ and the initial state $\mathfrak{A}$. The difference $\mathfrak{B} - \mathfrak{A}$ is the set of non-trivial updates that have to be applied to $\mathfrak{A}$ to get the state $\mathfrak{B}$.

Hence the essential difference between a rule call in Xasm and a rule call of a turbo ASM is that in Xasm the rule call means the *repeated* execution of the body, whereas for turbo ASMs it is just the one-time execution. In the presence of sequential composition (**seq**) the two approaches have the same expressive power, since the iterated execution of an ASM can be obtained as one step of an enclosing turbo ASM. From the logical point of view, the turbo ASM calls are simpler, since they just mean that the call has to be replaced by its body (see [15] for details).

References

[1] M. Anlauff and P. Kutter. Xasm Open Source. Web pages at `http://www.xasm.org/`, 2001. 261

[2] A. Blass and Y. Gurevich. Background, reserve, and Gandy machines. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, pages 1–17. Springer-Verlag, Lecture Notes in Computer Science 1862, 2000. 256

[3] A. Blass and Y. Gurevich. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic*, 2002. to appear. 245

[4] A. Blass and Y. Gurevich. Algorithms vs. machines. *The Logic in Computer Science Column, Bulletin of the European Association for Theoretical Computer Science*, 77:96–118, 2002. 246

[5] T. Bolognesi and E. Börger. Abstract State Processes. These Proceedings. 245

[6] E. Börger and T. Bolognesi. Remarks on Turbo ASMs for Functional Equations and Recursion Schemes. These Proceedings. 260

[7] E. Börger and J. Schmid. Composition and submachine concepts. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (CSL 2000)*, pages 41–60. Springer-Verlag, Lecture Notes in Computer Science 1862, 2000. 244, 248

[8] Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at `http://research.microsoft.com/foundations/AsmL/`, 2001. 261

[9] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1993. 256

[10] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, EECS Dept., University of Michigan, 1997. 256, 257

[11] Y. Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, 2000. 245

[12] Y. Gurevich and M. Spielmann. Recursive Abstract State Machines. *J. of Universal Computer Science*, 3(4):233–246, 1997. 246

[13] Y. Moschovakis. What is an algorithm? In B. Engquist and W. Schmid, editors, *Mathematics Unlimited — 2001 and beyond (Part II)*, pages 919–936. Springer-Verlag, 2001. 245

[14] J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`. 261

[15] R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. of Universal Computer Science*, 7(11):981–1006, 2001. 262

[16] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001. 251

Using Spin to Generate Tests from ASM Specifications

Angelo Gargantini[1], Elvinia Riccobene[2], and Salvatore Rinzivillo[3]

[1] CEA - Università di Catania (Italy)
[2] Dipartimento di Matematica e Informatica - Università di Catania (Italy)
[3] Dipartimento di Informatica - Università di Pisa (Italy)

Abstract. In this paper we introduce an algorithm to automatically encode an ASM specification in PROMELA, the language of the model checker Spin, and we present a method exploiting the counter example generation feature of Spin, to automatically generate from ASM specifications test sequences which accomplish a desired coverage. ASMs are used as test oracles to predict the expected outputs of units under test. A prototype tool that implements the proposed method is also presented. Experimental results in evaluating the method are reported. The experiments include test sequence generation, tests execution, code coverage measurement for a case study implemented in Java, and comparison with random tests generation. Benefits and limitations in using model checking are discussed.

1 Introduction

Along the process of software development, testing is of extreme importance to produce high-quality products. According to the recent NIST report [16], the cost of inadequate software testing infrastructure for the US economy is estimated to be $59.5 billions. Software needs to be tested to uncover development and coding errors, to assess its reliability and dependability, and to convince customers that the performance is acceptable. However, software testing is extremely costly and time-consuming. *Specification based testing* [10] offers an opportunity to significantly reduce the testing costs.

In specification based testing, a specification can be used as *test oracle* [13], i.e. as authoritative font of the expected system behavior, and as a means to assess correctness of implementations. Moreover, *test adequacy criteria* can be derived from a specification [18]. They determine if a test suite is adequate to test a software, whether enough testing has been performed or further tests are needed. A specification can also provide *selection criteria* of adequate test suites. Normally a selection criterion introduces some algorithms or techniques to actually generate test sequences from formal specifications.

In [8] we showed how to use an ASM as test oracle, we introduced and motivated several test criteria for ASMs, and we briefly explained how to use such test criteria to generate test suites exploiting the model checker SMV. In this paper we investigate further the use of model checkers for test suite generation.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 263–277, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Section 2 introduces the model checker Spin [11] and a novel algorithm to translate (a particular class of) ASMs in PROMELA, the language of Spin. This translation also allows using Spin to prove properties of ASM models, but formal verification is outside the scope of this paper. Section 3 explains the method of generating test sequences using model checkers and how, in particular, the model checker Spin is exploited to generate tests. Benefits and limitations in using model checkers for test generation are discussed. Section 4 briefly presents the tool we have developed. In Section 5, interesting experimental results in applying our method to real cases and to real code are presented. For the Safety Injection System (SIS) [5], test sequences are generated, and the coverage of the SIS Java code provided by these tests is measured through a code coverage analyzer. Moreover, these tests generated from the ASM specification are compared with test sequences randomly generated. Finally, we present the generation of test sequences for the OpenDoor example, overcoming some problems discussed in [9]. Section 6 concludes discussing related work and future directions.

2 Encoding ASMs in Spin

The model checker Spin [11] has been successfully used for software design and verification, and to trace logical design errors in distributed systems design. It uses a high level language, called PROMELA (PROcess MEta LAnguage), to specify systems. The tool checks the logical consistency of a specification and is able to prove system properties. It works *on-the-fly*, which means that it does not need to construct a global state graph as a prerequisite for the verification of any system properties.

Spin supports random, interactive and guided simulations, and both exhaustive and partial proof techniques. The tool is meant to scale smoothly with problem size, and is specifically designed to handle even very large problem sizes.

Spin can be used as a full LTL model checking system, supporting all correctness requirements expressible in linear time temporal logic, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Many of the latter properties can be expressed and verified using *assertions* instead of LTL.

A property is verified by creating an on-the-fly verifier program that can be compiled and then executed to eventually find a counterexample if the property is proved to be false. To optimize the verification runs, the tool exploits efficient partial order reduction techniques, and (optionally) BDD-like storage techniques.

Each PROMELA model has an `init` process representing the entry-point for the model simulation and verification. The other processes are instantiated and executed by this main process. All these processes are executed asynchronously. A model can contain only one `init` process.

Two processes can communicate through a *buffered channel*. Each channel has a fixed size and a tuple of types $T_1, \ldots, T_n$ for messages that can be received on that channel. Each message contains n fields, where the ith fields is

of type T_i. A process sends a message to a channel `cname` with the statement `cname!expr1,...,exprn`. A message is read from a channel `cname` with the instruction `cname?var1,...,varn`, where `var1`,..., `varn` are the variables where to store the actual values of the message. When a process reads a message from a channel, this message is removed from it. If a process tries to write on a full channel, then it is suspended until there is space for a new message. Analogously, if a process tries to read from an empty channel, it is suspended until a message arrives. *Rendezvous* channels are a special type of channels that permit synchronization between processes. These channels are zero-sized. When a process tries to write into a rendezvous channel, it is suspended until another process tries to read from it. Spin supports process communication also through *shared memory*.

In this section, we introduce a method to translate ASM specifications to PROMELA. We consider only a particular subset of sequential ASMs that we call ASM_0. ASM_0 specifications use only nullary dynamic functions (variables) and static functions that are those built-in functions of Spin (like +, &,). ASM_0s have only *finite integer*, *boolean* and *enumeration* domains, and do not allow the construct *choose* (non determinism is only in monitored variables). We assume ASM_0 specifications to be consistent with respect to the updates.

2.1 States

We show here how an ASM_0 state is encoded in PROMELA.

Universes. Integer and boolean domains of ASM_0 are simply mapped into PROMELA `int` and `bool` types, respectively. Enumerated domains can be modeled as integers through the definition of suitable constants. For example, an enumeration type $EnumName = Enum_0, \ldots, Enum_n$ is encoded as a sequence of constants declaration:

```
#define Enum_0 0
#define Enum_1 1
...
#define Enum_n n
```

Variables. ASM_0 dynamic nullary-functions are encoded as PROMELA variables. Each ASM_0 controlled variable is translated into a pair of variables, one yielding the *current value* and one yielding the *next value*. A similar method is used also in [14]. For example, for an integer variable *aVariable* we have:

```
int aVariable;
int aVariable_P;
```

where the suffix `_P` is used to distinguish the variable yielding the next value. The current value `aVariable` is used in guards and in expressions evaluation, while `aVariable_P` is updated by rules. After all rules are executed, each next value is copied in the current one, so to obtain the next state.

Monitored Variables Updating. An ASM_0 monitored variable does not need to be encoded as a pair of variables, because it is changed only by the environment and never updated by any rule. Monitored variables might have a completely random behavior, i.e. at each step they can take any value in their domain, or they may obey to some constraints (for example, the pressure in a boiler can change in a limited range from one state to the next one). ASM_0 allows to specify the interval and the maximum variation between two consecutive values for integer monitored variables. If a variable can have only values in an interval [a,b], we say that it is of type `int`$([a, b])$. Moreover, an integer variable may have a fixed *delta*, i.e. a maximum variation of value from the current state to the next one. When a *delta* is provided, we say the location is of type `int`$(interval, delta)$. If f is a monitored variable of type `int`$([min_f, max_f], \delta_f)$ and its current value is t, then in the next state the variable value will be in range $[max(t - \delta_f, min_f), min(t + \delta_f, max_f)]$. For example, let x be a variable of type `int`([0,200],5). Then the minimum value is 0, the maximum is 200; if the value of x in the current state is 120 then in the next state it must be in range $[115, 125]$. In Spin it is possible to use a non-deterministic statement to choose one of the values in the range. For example, the monitored location `aMonVariable` of type `int`([1, 10],2) is simulated as:

```
if
  :: skip
  :: (aMonVariable - 1 >=  1) -> aMonVariable = aMonVariable - 1;
  :: (aMonVariable + 1 <= 10) -> aMonVariable = aMonVariable + 1;
  :: (aMonVariable - 2 >=  1) -> aMonVariable = aMonVariable - 2;
  :: (aMonVariable + 2 <= 10) -> aMonVariable = aMonVariable + 2;
fi;
```

If the variable has no *delta*, then the next value is chosen randomly in the interval definition. For example, if the monitored location `aMonVariable` has type `int`([1, 10]) in PROMELA we have:

```
if
  :: aMonVariable = 1;
  :: aMonVariable = 2;
  ...
  :: aMonVariable = 10;
fi;
```

Variables of enumeration type can assume one of the possible values in type declaration. In Spin, a monitored variable `enumVariable` of type *EnumType* = *firstItem*, *secondItem*, *thirdItem*, is simulated as:

```
if
  :: enumVariable = firstItem;
  :: enumVariable = secondItem;
  :: enumVariable = thirdItem;
fi;
```

Constants. Constants are encoded with constant declarations.

2.2 Rules

ASM_0 rules are unfolded so to have only rules of the form

$$R_s = \texttt{if } G \texttt{ then } U \texttt{ else skip}$$

where U is a sequence of function updates. The unfolding method is explained in the following. Let $R = \texttt{if } G \texttt{ then } R_T \texttt{ else } R_E$ be a generic ASM_0 rule, where:

$$R_T = \begin{cases} \texttt{if } G_T^1 \texttt{ then } R_T^1 \\ \cdots \\ \texttt{if } G_T^n \texttt{ then } R_T^n \end{cases} \qquad R_E = \begin{cases} \texttt{if } G_E^1 \texttt{ then } R_E^1 \\ \cdots \\ \texttt{if } G_E^m \texttt{ then } R_E^m \end{cases}$$

The unfolding method showed in the following schema (similar to that presented in [4]) allows to translate R in a set of simpler rules of R_s form:

$$R = \texttt{if } G \texttt{ then } R_T \texttt{ else } R_E \;\rightarrow \begin{cases} R_T^1 = \texttt{if } G \wedge G_T^1 \texttt{ then } R_T^1 \\ \cdots \\ R_T^n = \texttt{if } G \wedge G_T^n \texttt{ then } R_T^n \\ R_E^1 = \texttt{if } \neg G \wedge G_E^1 \texttt{ then } R_E^1 \\ \cdots \\ R_E^m = \texttt{if } \neg G \wedge G_E^m \texttt{ then } R_E^m \end{cases}$$

In the following we present two methods for PROMELA encoding of rules like R_s.

Flat Translation. The first method uses only one process `init` to execute the model. For each rule we introduce an `if ... fi` statement. So each rule $R_i = \texttt{if } G_i \texttt{ then } U_i$ is translated into:

```
if
  :: G_i -> U_i;
  :: else -> skip;
fi;
```

The final model for the model checker is shown in Figure 1. After enumeration, variables and constant declarations, the `init` process begins. It contains an unique `do` cycle that performs monitored variable updating, executes the rules and updates current variable values. Each `do` cycle corresponds to a step of the original ASM_0.

Chan Translation. This method uses a process for each rule, so that all the rules are executed asynchronously. The main process has the task of synchronizing the execution of all the rule processes. This synchronization is possible through *rendezvous* channels. Each rule process has one channel and the rule execution is suspended until a message arrives in the channel. For a rule $R_i =$ `if` G_i `then` U_i, we have in PROMELA:

```
mtype={start}
chan r_i_chan = [0] of {mtype}

proctype r_i(){
  do :: atomic{
          r_i_chan?start;
          if
            :: G_i-> U_i;
            :: else -> skip:
          fi
        }
  od
}
```

The statement `atomic` ensures that the statements in the brackets are executed atomically, without interruptions. The rule body is encoded in the same way as in the *flat* method. The statement `r_i_chan?start` tries to read a message `start` from the channel `r_i_chan`. The keyword `mtype` define a special type for messages in channels.

In Figure 1 is shown a schema of a Spin model obtained with chan translation. The `init` process starts all the rule processes. Inside the main `do` cycle, for each rule `r_i`, the statement `r_i!start` let the rule `r_i` fire:

```
/* init function */
init{               /* main processing loop */
  run r_i();        /* start rule processes */
  do ::
    ...             /* Simulation of monitored variables */
  atomic{           /* Rules execution */
     r_i!start;     /* Fire rule r_i */
     ...
     ...            /* Locations update */
  } od
}
```

Comparison. We have compared the two translation methods for the specification of the case study SIS [5]. Executing PROMELA specifications obtained

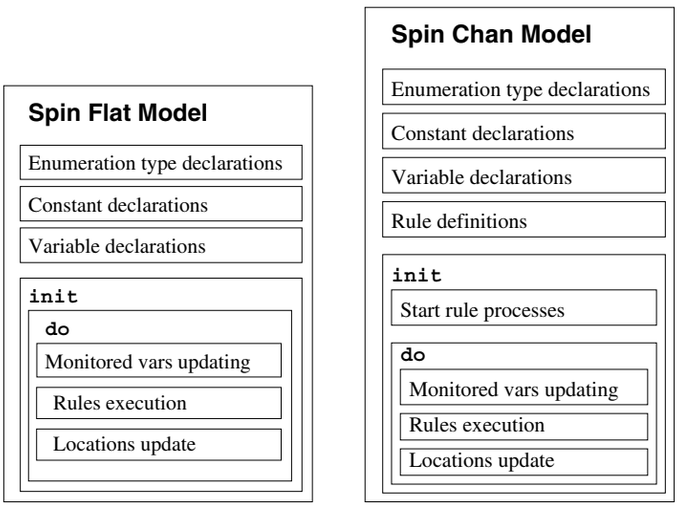


Fig. 1. Translation structure

by chan translation requires much more memory than executing those obtained by flat translation. However, chan models run slightly faster than flat models.

Although we assume ASM_0s to be sequential, the proposed chan translation method could be easily extended to deal with distributed ASMs.

3 Automatic Generation of Test Sequences from ASM Specifications

In this Section we briefly review some fundamental definitions and concepts originally presented in [8]. Moreover, we introduce a new coverage criteria, explain the use of Spin for tests generation and discuss its benefits and limitations.

Testing Criteria for ASMs. In [8], several structural coverage criteria for ASM specifications are defined. They are: *basic rule* coverage, *rule update* coverage, *parallel rule* coverage, and *MCDC*. Definitions and motivations for these criteria are not reported here for the sake of brevity; the reader can refer to [8] for more details. These criteria represent the coverage of particular behaviors of the specified system, and besides them, new criteria can be introduced to achieve other coverage. In this section, we define and explain the *full guard coverage* criterion, which is an adaptation of the full predicate coverage criterion defined in [12].

Definition 1. *A test suite T satisfies the Full Guard Coverage criterion if for each rule guard G, T includes tests that cause each atomic condition c in G to*

result in a pair of outcomes where the value of G is directly correlated with the value of c.

In this definition, "directly correlated" means that c controls the value of G, that is, one of the following two situations occurs: either c and G have the same value (c is true implies G is true and c is false implies G is false), or c and G have opposite values (c is true implies G is false and c is false implies G is true). This criterion is very similar to the MCDC, but when testing a particular atomic condition, it does not require to hold fixed all other atomic conditions. For example, let the guard G be $a \wedge (b \vee c)$. Consider the atomic condition a. In order to test a in G according to the full guard coverage, we need two tests: a test t_1 with a true and $b \vee c$ true, and a second test t_2 with a false and $b \vee c$ true. Therefore, $t_1 = \{a, b, \neg c\}$ and $t_2 = \{\neg a, \neg b, c\}$ are sufficient to cover a in G according the full guard coverage, but they are not adequate to cover a according to the MCDC. For the MCDC, indeed, b and c must have the same value in t_1 and in t_2.

Test Predicates. We formally define a coverage criterion using a set of logical predicates, called *test predicates*. A test predicate is a formula over the state and determines if a particular testing goal is reached (e.g. a particular condition or a particular event of the specification is covered). A test suite T satisfies a coverage C if each test predicate of C is true in at least one state of a test sequence of T.

For example for the full guard coverage of a in $G = a \wedge (b \vee c)$, we introduce two test predicates: $tp_1 = a \wedge (b \vee c)$ and $tp_2 = \neg a \wedge (b \vee c)$. Test predicate tp_1 implies G, while tp_2 implies $\neg G$, and the value of G depends only on the value of a. We search tests which contain a state where tp_1 is true and a state where tp_2 is true.

Automatic Test Sequence Generation Using Model Checking. The method proposed for automatic test generation is based on the model checker Spin. This technique is explained in [8], where the model checker SMV is used instead of Spin, and it is here only briefly sketched.

To automatically generate test sequences, we exploit the model checker capability to discover counter examples for properties that do not hold in the model. The method consists in several steps. First, we compute for the specification and a desired coverage criteria the test predicates set $\{tp_i\}$. Second, we encode the ASM specification in PROMELA, the language of Spin, following the algorithm described in Section 2. Third, we compute for each test predicate tp_i the test sequence that covers tp_i by trying to prove with Spin the *trap property* $\neg tp_i$. If Spin finds a state s where tp_i is true, it stops and prints as counter example the state sequence leading to s. This sequence is the test covering tp_i.

Infeasible Tests. Spin always terminates and one of the following three situations occurs. The best case is when Spin stops finding that the trap property is false, and, therefore, the counter example to cover the test predicate is generated.

The second case happens when Spin explores the whole state space without finding any state where the trap property is false, and, therefore, Spin proves $\neg tp_i$. In this case, we say that the test predicate is *infeasible* or *not coverable*. Infeasible test predicates are to be ignored for testing purposes. We define *feasible* [18] versions of our adequacy criteria requiring that a test suite T satisfies a coverage C if each *feasible* test predicate of C is true in at least one state of a test sequence of T. In the following, we always refer to feasible versions of coverage criteria.

In the third case, Spin terminates without exploring the whole state space and without finding a violation of the trap property, and, therefore, without producing any counter example (generally because of the state explosion problem). In this case, the user does not know if either the trap property is true (i.e. the test is infeasible) but too difficult to prove, or it is false but a counter example is too hard to find. When this case happens, our method simply warns the tester that the test predicate has not been covered, but it might be feasible. The use of abstraction to reduce the likelihood of such cases is under investigation. The possible failure of our method should not surprise: the problem of finding a test that covers a particular predicate is undecidable [17]. Nevertheless, in our experience the third case is quite rare: for our case study it never happened.

Model Checking Limits. Model checking applies only to finite models. Therefore, our method works for ASM specifications having variables and functions with finite domains. However, this limitation does not preclude the application of our approach to models with infinite domains, because an ASM A with finite domains can be extracted from an ASM B with infinite domains, such that a test generated for A is still a valid test for B. The problem of abstracting models with finite domains from models with infinite domains such that some behaviors are preserved, is under investigation.

Moreover, since model checkers perform exhaustive state space (possibly symbolic) exploration, they fail when the state space becomes too big and intractable. This problem is known as *state explosion problem* and represents the major limitation in using model checkers. Note, however, that we use the model checker not as a prover of properties we expect to be true, but to find counter examples for trap properties we expect to be false. Therefore, our method does generally require a limited search in the state space and not an exhaustive state exploration.

Model Checking Benefits. Besides its limits, model checking offers several benefits. Existing model checkers, and Spin in particular, adopt sophisticated techniques to compute and explore the state space, and to find property violations. They represent a state and the state space in a very efficient way using state enumeration, hashing techniques and symbolic representations. Moreover, model checkers explore the state space using practical heuristics and other techniques. For example, Spin uses partial order reduction methods and on-the-fly state exploration. For these reasons, we have preferred existing model checkers

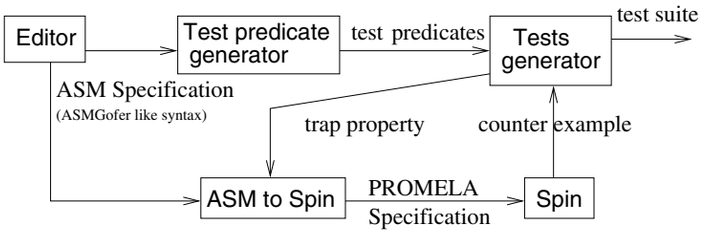


Fig. 2. Tool Architecture

instead of developing our own tools and algorithms for state space exploration. Moreover, the complete automaticity of Spin allows to compute test sequences from PROMELA specifications without any human interaction.

4 A Tool for Automatic Generation of Test Sequences

We have developed a prototype tool[1] that applying the method presented in the previous section, automatically produces a complete test suite for several coverage criteria. The tool architecture is depicted in Figure 2. The Editor allows the designer to edit an ASM_0 specification. The ASM_0 grammar is a subset of the AsmGofer [15] language, extended to allow the definition of intervals and delta as explained in Section 2.1. The Test predicate generator computes the test predicates for the following coverage criteria: basic rule, complete rule, rule update, rule guard, full guard coverage. The Tests generator controls the generation of the test suite: (a) it takes each test predicate selected by the user and generates the corresponding trap property; (b) it calls the ASM_to_Spin component to translate the specification, together with the trap property to PROMELA; (c) it runs Spin; (d) it takes the counter example generated by Spin (if any), and decides, according to the user preferences, to stop or to continue test generation till each required coverage of test predicates has been satisfied. At the end, a set of tests, or test suite, is produced. The test suite can be displayed and saved into a text file.

Remark 1. During the test generation for a test predicate, it also possible to check if other test predicates $p_1, ..., p_n$ are covered, adding to the PROMELA specification a statement like:

```
if
  :: P_1 -> printf("Covered: caseP_1");
fi;
  ...
if
  :: P_n -> printf("Covered: caseP_n");
fi;
```

[1] The tool is available at: `http://www.dmi.unict.it/garganti/atgt/`.

The search for common coverage can reduce dramatically the number of tests to generate.

5 Experimental Results

The main goal of the experiments we conduct, is to provide evidence that our method is effective. In particular, we tackle the fundamental question about the quality of the tests our method generates, i.e. whether they are suitable to test programs developed from ASM specifications. Indeed, our test sequences, as always in specification based testing, are generated regardless the real code of the implementation, but the final goal of testing is discovering faults in code and producing quality software.

First, we generate a complete test suite for the Safety Injection System (SIS) specification [5], for which the Java code is available. The complete description of SIS and its ASM specification can be found in [8]. Table 1 reports our results in test generation.

In the first and second column, respectively, Table 1 reports the coverage and the number of test predicates to achieve it. We start generating a test sequence for each test predicate, but since the tool has the capability to detect if a test predicate is covered by a test sequence already generated (as explained in Remark 1), the generation of one test for each test predicate is not necessary and we avoid to generate test sequences for test predicates already covered. This fact explains why the number of generated tests shown in the third column is less than the number of test predicates in the second column. Moreover, if a test sequence t_1 covers a set of test predicates S_1 and another test sequence t_2 covers a set of test predicates S_2, and S_1 is a subset of S_2, then we consider t_1 useless and we do not include it in the test suite. For this reason, the number of useful tests shown in the forth column is less than the number of the generated tests. The fifth column reports the name of the tests and in the sixth column their length as number of states. Note that we discover that one test predicate is not feasible, i.e., as explained in Section 3, there is no test sequence that can cover it. In the end, the test suite for the basic rule coverage is compound of 3 test sequences

Table 1. Test generation for SIS

Coverage	# test pred.	# generated	# useful	Test name	Test length
Basic Rule	18	4	3	BR_R7_1 BR_NEG_R7_1 BR_R3 BR_R4	1 32 (useless) 910 1034
Complete Rule	9	0	0		
Rule update	11	2	2	UR_R4_2 UR_R2_2	not feasible 1000
Full predicate	38	0	0		
Rule guard	9	0	0		

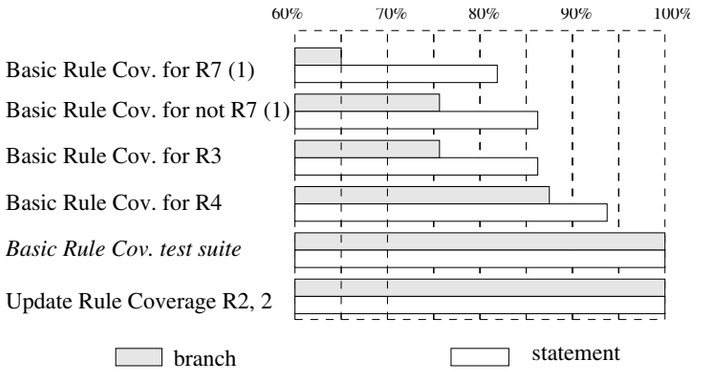


Fig. 3. Code coverage results

(BR_R7_1, BR_R3, and BR_R4), while the complete test suite to achieve every coverage is compound of 4 tests (previous three tests and UR_R2_2).

Second, we apply the test suite to the Java code for SIS. We modify the SIS code in such a way that the code reads the monitored values from a specific file instead of from the real environment. Then, we evaluate the code coverage using an automatic coverage tool, JCover[2], that is capable of computing the branch and statement coverage of a given Java program. Results are reported in Figure 3. Branch and statement coverage are achieved already by the test suite for the basic rule coverage. This experiment suggests that a programmer should generally achieve the statement and branch coverage by running code with test suites for the basic rule coverage. Intuitively, each branch in the code corresponds to some rule guard in the specification and, therefore, testing specification guards as true and false, corresponds to testing branches in the code. Moreover, we claim that other coverage criteria (for example Update Coverage Criteria) can guarantee a better code coverage than branch coverage, and that statement and branch coverage are coarser than our criteria. In the future, we plan to evaluate the code coverage with more power tools than JCover (for example, LDBA Testbed©).

To judge further the effectiveness of the generated test suite, we compare it with several randomly generated tests. In generating random tests, we make monitored variables randomly change, taking into account only environmental constraints on monitored variables, i.e. their interval and the delta between their values in two consecutive states. Results are reported in the Figure 4.

In random generation, only one test with length 10^5 states can achieve the statement and branch coverage, while in specification based testing, three tests (BR_R7_1, BR_R3, BR_R4) with a total length of about 2000 states are enough. This observation suggests that random test generation might not be able to

[2] JCover^{TM} is a Java Code Coverage Analyzer developed by Man Machine Systems (`www.mmsindia.com`)

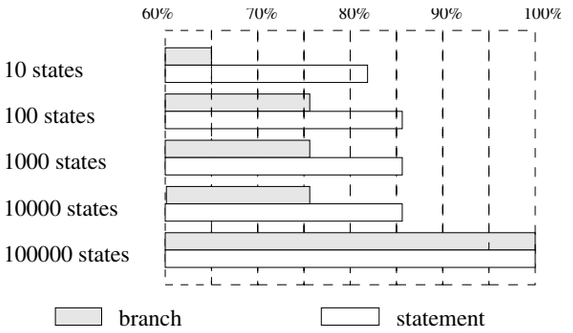


Fig. 4. Random generation coverage results

provide the same coverage as the specification based testing, and that, in case that both achieve the same coverage, our method can lead to test sequences shorter than the random tests. A thorough comparison between specification based and random testing is still subject of many research papers and it is clearly out of the scope of this work.

As final experiment, we generate a test suite for the specification of a UPnP door controller, briefly OpenDoor, as presented in [9]. In that specification a controlled variable `counter` is initially equal to 0 and is incremented by the rule `Close` till it reaches a `max` value (representing the number of CD slots) and then it restarts from 0. The part of the rule `Close` that causes some problem in tests generation, is

```
If open and counter = max
then          open := false
              counter :=0
```

In [9] the proposed algorithm *never discovers the reachable hyperstate where the value of counter equals max* [9], i.e. it never discovers when the guard `open and counter = max` becomes true. This problem is called "non discovery problem" in that report. Note that discovering whether a state is reachable or not is undecidable, and finding a trace to cover a particular state is also undecidable, so there's no algorithm that can definitively solve the non discovery problem. However, our method is able to find a test sequence for the basic rule coverage of `Close` even for great values of `max`. Setting `max` = 100000, Spin takes 1 minute and 23 seconds[3] to generate the counter example to cover the critical guard of `Close`.

6 Related Work and Future Directions

Test generation using model checkers has been proposed in several papers. In [6] the authors present a method to generate test sequences using Spin. Their ap-

[3] Spin running on a PC Pentium III with 866 Mhz and 256 Mb RAM.

proach exploits the counter example generation of Spin, but they do not introduce coverage criteria, they require the designer to translate his/her specification in PROMELA and to insert testing goals by hand, suitably modifying the original specification. In [2] the model checker SMV is used in combination to the mutation analysis to generate tests from mutated specifications. The coverage is measured in terms of the number of incorrect mutations that can be detected (this criterion is called *mutation adequacy*). In [7] tests are generated using model checkers (both SMV and Spin) from SCR specifications to achieve a coverage similar to the branch coverage for programs, or to cover particular system requirements.

To the best of our knowledge, the only method for generating test suites from ASM specifications is that recently developed by the Microsoft group in Redmond [9]. In order to find a test suite, they extract a finite state machine from ASM specifications and then use test generation techniques for FSMs. The problem of reducing ASMs to FSMs is undecidable. To assess the quality of the testing activity, they still need to define some coverage criteria.

The possibility to automatically encode ASM specifications in PROMELA, the automatic generation of trap properties from coverage criteria, together with the Spin automatic computation of test sequences from PROMELA specifications, allow us to develop a method to generate test sequences from ASMs in a completely automatic way. In the future, we plan to introduce some abstraction techniques that allow the extraction of models with finite domains from models with infinite domains such that some behaviors are preserved, and that reduce the likelihood of the state explosion problem (as in [3]). We are also currently working to improve our tool and to support tests generation from ASML specifications [1].

References

[1] AsmL. `http://research.microsoft.com/fse/asml/`. 276

[2] Paul Ammann, Paul Black, and William Majursk. Using model checking to generate tests from specifications. In *Proceedings of 2nd IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, Brisbane, Australia, December 1998. 276

[3] Ramesh Bharadwaj and Constance L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering*, 6(1):37–68, January 1999. 276

[4] Giuseppe Del Castillo and Kirsten Winter. Model checking support for the ASM high-level language. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 331–346, 2000. 267

[5] P.-J. Courtois and David L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993. 264, 268, 273

[6] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In E. Brinksma, editor, *TACAS'97, Lecture Notes in Computer Science 1217*, pages 384–398. Springer, 1997. 275

[7] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engeneering*, volume 1687 of *LNCS*, sep 1999. 276

[8] Angelo Gargantini and Elvinia Riccobene. Asm-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11), 2001. 263, 269, 270, 273

[9] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Conformance testing with abstract state machines. Technical Report MSR-TR-2001-97, Microsoft Reasearch, October 2001. 264, 275, 276

[10] R. Hierons and J. Derrick. Special issue on specification-based testing. *Software testing, verification & reliability (STVR)*, 10(4):201–262, December 2000. 263

[11] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 264

[12] A. Jefferson Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, 1999. 269

[13] D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proceedings of the 14th International Conference on Software Engineering*, pages 105–118. Springer, May 1992. 263

[14] J. Schmid. Compiling abstract state machines to C++. *Journal of Universal Computer Science*, 7(11), 2001. 265

[15] J. Schmid. *Introduction to AsmGofer*, 2001. 272

[16] Gregory Tassey. The economic impacts inadequateinfrastructure software testing. Technical Report Planning Report 02-3, National Institute ofStandards and Technology, May 2002. 263

[17] Elaine J. Weyuker. Translatability and decidability questions for restricted classes of program schemas. *SIAM Journal on Computing*, 8(4):587–598, 1979. 271

[18] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit text coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997. 263, 271

Interfacing ASM with the MDG Tool

Amjad Gawanmeh[1], Sofiène Tahar[1], and Kirsten Winter[2]

[1] Concordia University, Montreal, Quebec, H3G 1M8 Canada
{amjad,tahar}@ece.concordia.ca
[2] SVRC, University of Queensland, Australia
kirsten@svrc.uq.edu.au

Abstract. In this paper we describe an approach to interface Abstract State Machines (ASM) with Multiway Decision Graphs (MDG) to enable tool support for the formal verification of ASM descriptions. ASM is a specification method for software and hardware providing a powerful means of modeling various kinds of systems. MDGs are decision diagrams based on abstract representation of data and are used primarily for modeling hardware systems. The notions of ASM and MDG are hence closely related to each other, making it appealing to link these two concepts. The proposed interface between ASM and MDG uses two steps: first, the ASM model is transformed into a flat, simple transition system as an intermediate model. Second, this intermediate model is transformed into the syntax of the input language of the MDG tool, MDG-HDL. We have successfully applied this transformation scheme on a case study, the Island Tunnel Controller, where we automatically generated the corresponding MDG-HDL models from ASM specifications.

1 Introduction

With the increasing reliance on digital systems, errors in their design can cause failures, resulting in the loss of time, money, and a long design cycle. Large amounts of effort are required to correct an error, especially when the error is discovered late in the design process. For these reasons, we need approaches that enable us to discover errors and validate designs as early as possible. Conventionally, simulation has been the main debugging technique. However, due to the increasing complexity of digital systems, it is becoming impossible to simulate large designs adequately. Therefore, there has been a recent surge of interest in formal verification and tool support for this task, such as theorem proving, combinational and sequential equivalence checking, and model checking [11]. These approaches vary in the degree of interaction that is required by the user. Of particular interest are those tools that run automatically and do not require any special knowledge about the formal techniques that are applied. Equivalence and model checkers belong to this category.

ASM (Abstract State Machines) [8] is a specification method for software and hardware modeling and provide a powerful means of modeling various kinds of systems. MDGs (Multiway Decision Graphs) [3] are decision diagrams based

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 278–292, 2003.
© Springer-Verlag Berlin Heidelberg 2003

on abstract representation of data and are used for modeling hardware systems in first place. The MDG tool provides equivalence checking and model checking applications based on MDG. The given modeling language is the hardware description language MDG-HDL [20]. In this work, we propose to build an interface from ASM to the MDG tool in order to enable the formal verification of ASM descriptions using automated tools. This interface serves two purposes, depending on the perspective: firstly, it provides tool support for automated verification of ASM models. Secondly, it provides the MDG tool with a high-level modeling language, namely ASM, which allows us to model a wide range of applications in an understandable and succinct manner.

Our work is based on earlier work on an interface from ASM to the model checker SMV [6, 17] and also from ASM to MDG [17]. In contrast to the SMV model checker, the MDG tool provides a useful means for representing abstract models containing uninterpreted functions, where SMV supports neither abstract data types nor uninterpreted functions. This allows model checking on an abstract level at which the state explosion problem can in some cases be avoided. Although the latter approach in [17] already provides support for abstraction by exploiting the MDG data structure, it does not provide an interface to the actual MDG tool. Our work provides a tool that transforms ASM-SL, the modeling language of the ASM Workbench (ASM-WB) [5], into MDG-HDL code which can be input into the given MDG tool.

Other work on model checking ASM specifications include the complexity analysis of by Spielmann [14] and the work on real-time systems by Beauquier and Slissenko [2], which provide very useful theoretical results. Moreover, an interface to the model checker SVE is suggested by Plonka [13]. These results are complemented by our work since the MDG tool facilitates the handling of functions over abstract domains and ranges. ¿From a more general perspective, the work described by Shankar [15] and Katz and Grumberg [9] is also related in that they provide a very general tool framework comprising a general intermediate language which allows one to interface a high-level modeling language with a variety of tools. In [10], Kort *et al.* propose a hybrid verification tool linking MDG and the HOL theorem prover, making it possible to delegate the verification of HOL subgoals to the MDG tool for automatic proof. Our work results in an ASM-specific solution which extends the interface framework of [6, 17] as well as [10] with another tool interface.

2 Abstract State Machines

Abstract State Machines (ASM) [8] is a specification method for software and hardware modeling. The system is modeled by a set of states and transition rules which specifies the behavior of the system. Transition rules specify possible state changes according to a certain condition. The notation of ASM is efficient for modeling a wide range of systems and algorithms as the number of case studies demonstrates [8].

An ASM model consists of states and transition rules. States are given as many sorted first-order structures, and are usually described in terms of functions. The ASM notation includes *static functions* and *dynamic functions*. Static functions have the same interpretation in every state, while dynamic functions may change their interpretation during a run. There are also *external functions* which cannot be changed by the system itself, but by the outer environment.

Dynamic as well as static components of the system can be described within the same ASM model. We can also have behavioral (specification) as well as a structural description (implementation) for the same system. This is the typical way for modeling hardware designs. A behavioral description is a higher level model of the system, we use *if-then-else* rules and *dynamic functions* to describe the system behavior. On the other hand, a structural description is a lower-level model in which we use *static functions* to define hardware primitives, such as logic gates. From these primitives we build a hierarchical or modular structure of the system.

The ASM Workbench (ASM-WB) [5] provides a number of basic functions including: parsing, type checking, pretty printing and evaluation of terms and rules. It supports computer aided specification, analysis and validation of ASM models. The ASM-WB supports the ASM specification language (ASM-SL) which is a strongly typed version of ASM. The ASM-WB is designed as an extensible tool, to which different tool interfaces can be added. One interface with the SMV model checker was suggested and implemented by [6]. Since our work is built on preliminary work [17], we can furthermore exploit the notion of *abstract types*. This concept provides a simple means for abstraction in order to abstract from infinite data types. This feature can be essential when applying automated verification techniques like model checking.

3 Multiway Decision Graphs

Multiway Decision Graphs (MDGs) [4] is a relatively new class of decision diagrams which subsumes the traditional ROBDDs (Reduced Order Binary Decision Diagrams) [11] while allowing abstract data sorts and uninterpreted function symbols. MDGs are based on a subset of many-sorted first-order logic, with a distinction between *abstract* and *concrete* sorts (including the Boolean sort). Concrete sorts have *enumeration* while abstract sorts do not. The enumeration of a concrete sort is a set of distinct constants occurring in the enumeration and referred to as *individual constants*. Constants of abstract sort are called *generic constants* and could be viewed as 0-ary function symbols. The distinction between abstract and concrete sorts leads to a distinction between three kinds of function symbols. Let f be a function symbol of type $\alpha_1 \times \alpha_2 \times \ldots \times \alpha_n \rightarrow \alpha_{n+1}$. If α_{n+1} is an abstract sort, then f is an *abstract function symbol.* If all the $\alpha_1 \ldots \alpha_n$ are concrete, then f is a concrete function symbol. If α_{n+1} is concrete while at least one of the $\alpha_1 \ldots \alpha_n$ is abstract, then f is referred to as a *cross-operator.* Concrete function symbols must have explicit definition; they can be eliminated

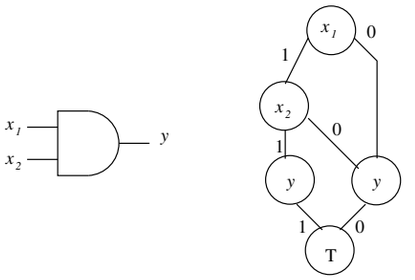


Fig. 1. MDG for an AND gate

and do not appear in MDG. Abstract function symbols and cross-operators are *uninterpreted.*

An MDG is a finite, directed acyclic graph (DAG). An internal node of an MDG can be a variable of concrete sort with its edge labels being the individual constants in the enumeration of the sort; or it can be a variable of abstract sort and its edges are labelled by abstract terms of the same sort; or it can be a cross-term (whose function symbol is a cross-operator). An MDG may only have one leaf node denoted as **T**, which means all paths in an MDG are true formulae. Thus, MDGs essentially represent relations rather than functions. MDGs can also represent sets of states.

3.1 System Modeling Using MDG

Logic gates can be represented by MDGs similarly to ROBDDs, because all inputs and outputs are of Boolean type. Figure 1 shows the MDG for an AND gate for a given variable order. A design description on the register transfer level (RTL), however, involves the use of more complex functions and data inputs that go beyond the capacity of ROBDDs. For example, Figure 2 shows the MDG of an arithmetic logic unit (ALU), where *op* is a concrete variable with enumeration sort {0,1,3,4}, *x1, x2* and *y* are abstract variables, *zero* is a generic (abstract) constant of the same sort, and *sub, add* and *inc* are uninterpreted functions.

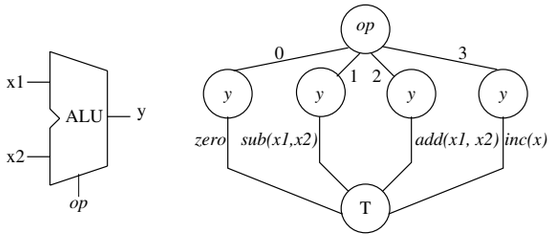


Fig. 2. MDG for an ALU

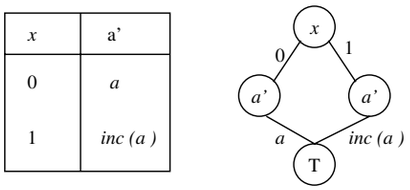


Fig. 3. Table and MDG for a simple behavioral state machine

To describe hardware circuits, the MDG tool provides a Prolog-style hardware description language called MDG-HDL [20]. It allows the use of abstract as well as concrete variables for representing data operations. A circuit can be described on the structural (implementation) or behavioral (specification) level. Often models on both levels of abstraction are given and shown to have equivalent behavior (e.g., by means of equivalence checking). A structural description is a collection of components connected by signals that can be of abstract or concrete type. MDG-HDL includes a library of predefined components that represent logic gates such as AND, OR, multiplexers, registers, drivers, etc. There is also a component that represents functions as a black box called *transform*. It is used for uninterpreted functions, or cross-terms.

The behavioral description is represented by means of abstract descriptions of state machines [1] defined in MDG-HDL in terms of *tables*. An MDG table is similar to a truth table, but it allows first-order terms as entries in addition to concrete variables. Tables usually describe the transition, the output relation, or the combinational functionality of the system. They contain a list of rows where the first row contains variables and cross-terms. Variables must be concrete except for the last element which can be abstract. This last element provides the resulting value of the function or transition. All other rows except the last one must contain individual constants in the enumeration of their corresponding variable sort, or the "*" which symbolizes a "don't care" value. The last element can be a constant value or a first-order term. Figure 3 shows a tabular description of a simple state machine, with its MDG representation, where x is a Boolean input, a is an abstract state variable and a' is its next state variable. It performs *inc* operation when $x = 1$, where *inc* is an uninterpreted function symbol.

3.2 Verification Using the MDG Tool

For the formal verification of finite state systems (machines), the MDG tool provides a set of application procedures for combinational and sequential equivalence checking [4], invariant checking [4] and model checking [18]. The MDG

[1] In the MDG literature [3] such a finite state machine (FSM) is called abstract state machine (ASM) which is obtained by letting some data input, state or output variables of an FSM be of abstract sort, and the data operations be uninterpreted function symbols.

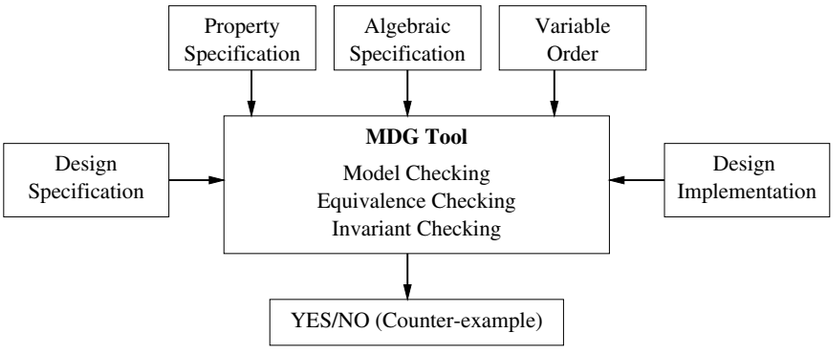


Fig. 4. MDG verification tool

tool has been used to verify a number of non-trivial systems such as communication switches and protocols [1, 3, 16, 19, 21].

Figure 4 summarizes the MDG tool applications. In order to verify designs with this tool, we need first to specify the design in MDG-HDL in terms of a behavioral and/or a structural description (design specification and design implementation in Figure 4). Moreover, an algebraic specification is to be given to declare sorts, function types, and generic constants that are used in the MDG-HDL description. Rewrite rules that are needed to interpret function symbols should be provided here as well. Like for ROBDDs, a symbol order according to which the MDG is built should be provided by the user. This symbol order can affect critically the size of the generated MDG. While the current version of MDG uses manual static ordering, a newer version will be released soon including automatic dynamic ordering [7].

4 Interfacing ASM with the MDG Tool

In order to provide a generic interface for the ASM-WB with different tools, ASM models are automatically translated into the ASM-IL [17]. An ASM-IL representation is a flat, simple transition system, which means that all nested rules in the ASM model are flattened and all complex data structures are unfolded. Based on this ASM-IL, we build an interface to the MDG tool. The disadvantage of a flattened representation in ASM-IL, however, is the fact that it does not preserve the structure of the original ASM model, because it provides no modular or hierarchical descriptions.

In ASM-IL, (*locations* are identified with state variables by mapping each location to a unique variable name. (*Guards* are mapped into simple Boolean terms. Thus, an ASM model is represented by a set of guarded updates in a triplet form (*loc, guard, val*). All nested rules are flattened then mapped into simple guarded updates using a simplification function. Each term that occurs in an ASM rule is simplified until the result contains only constants, locations and

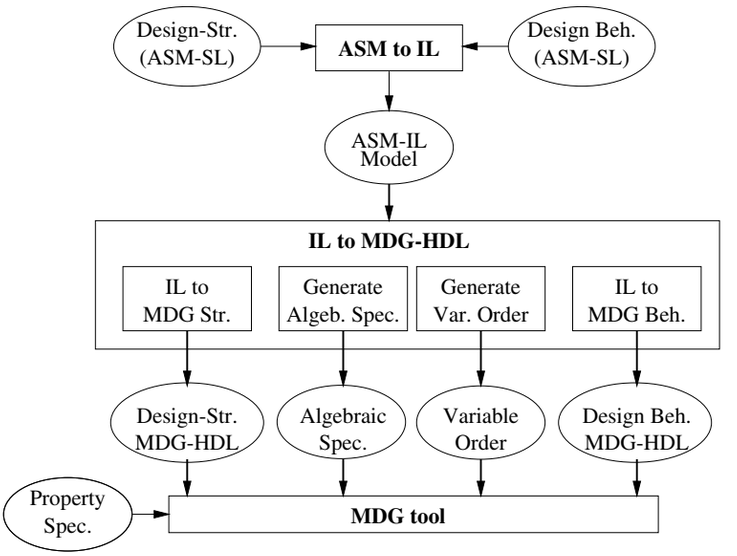


Fig. 5. ASM-MDG interface

variables. Abstract functions and cross-operators are left uninterpreted in ASM-IL. Only cross-operators that match one of the standard relational operators are mapped into a cross term.

Starting from the ASM-IL language, we built our interface, as shown in Figure 5, in four steps. Each step automatically generates one of the tool inputs: MDG-HDL design behavior and/or MDG-HDL design structure, algebraic specification, and symbol (variable) order. These steps are described in the subsections to follow. The interface was implemented and then tested with a case study.

4.1 MDG-HDL Structural Description

To build an MDG-HDL structural description from an ASM-IL model, we map locations, guards, and values into MDG-HDL components preserving the functionality of the model. Uninterpreted functions and abstract sorts in the original model are left unchanged.

Locations. In an ASM-IL representation each location is associated with a set of guarded updates, each consisting of a Boolean guard and an update value. The whole expression evaluates into one value, $value_i$, which is the next state value of the dynamic location, if there is at least one guard satisfied. Otherwise, the value of the location will be the same in the next state. Figure 6 shows the interpretation of guarded updates.

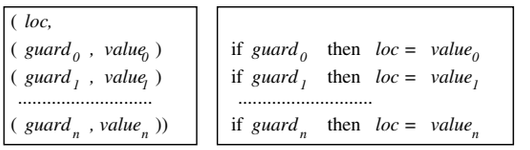


Fig. 6. ASM-IL guarded updates

First, each location is mapped into a state component in MDG-HDL. Since locations are considered state variables, we represent locations as registers. A register has two signals, an input and an output signal. Each location name is mapped into a signal that is connected to the register's output. The result value of the location is mapped to a signal that is connected to the register's input. This will generate a state machine (sequential circuit) in which the number of state variables is equal to the number of updated locations in the model.

For guards and values, we build a set of MDG-HDL components that are interconnected with signals that evaluate to the next state value of the location: Each pair (guard, value) is mapped into a multiplexer where the guard is the control and the value is one input. We connect these multiplexers together in a hierarchical way as shown in Figure 7. This output is connected into the input of the state element representing the location. The location is fed back into the last multiplexer in the hierarchy to represent the case in which no guard is satisfied.

Values. Values can be locations, constants, or any variable. If a value is a location or a variable, we map it directly into a signal with the same type. Constants are mapped into an MDG-HDL component called *constant_signal*, that is a component with an individual value from the enumeration of the concrete data sort or a generic constant in case of an abstract type. For Boolean types we use two default components that have always the values of 0 and 1.

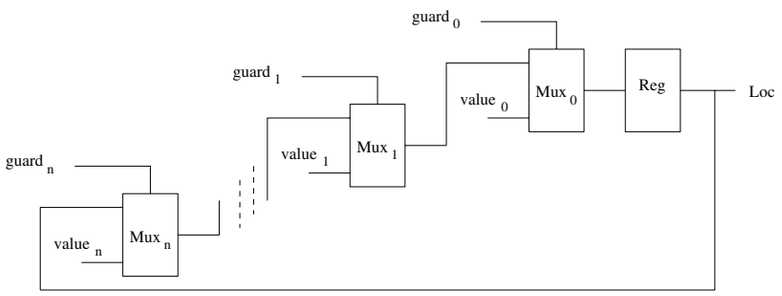


Fig. 7. Mapping ASM-IL expression for one location into MDG-HDL

Guards. A guard is a Boolean ASM-IL expression. It might contain concrete functions, uninterpreted functions, or cross-terms. Concrete functions can be default Boolean operators or any other function. We map these operators into MDG-HDL components that perform the same functionality. We apply the mapping function on each rule by creating an MDG-HDL component, then mapping the same function again on its parameters until we get a constant value, or a variable.

All default binary operators are mapped into MDG-HDL logic gates. An equality expression for a variable and the value **True** are simply mapped into a signal with the variable name. Equality expressions for a variable and the value **False** are mapped into the corresponding negated MDG-HDL component, **NOT**. Relational operators, such as >, >=, <, <=, etc., are mapped into a *transform* component that can be viewed as a black-box. All other cross-terms, abstract functions, and uninterpreted functions are also mapped into *transform* components.

4.2 MDG-HDL Behavioral Description

MDG-HDL behavioral descriptions are represented by a *table* similar to a truth table. For each location in the ASM model, we generate one table. The first row of the table contains all variables in the model and any cross-term or function that occur in the ASM-IL guarded update expression of that location. The last element is the location itself. Then we treat the list of (*guard, value*) pairs one by one (see Figure 8). An expression with one variable in the guard is mapped into one row with all other variables are set to the "don't care" ("*") symbol. A conjunction is mapped into one row with each variable or cross-term is assigned its value (val_i), or "don't care" if it does not occur in the expressions. The result *value* is assigned to the last element in the row, which gives the valuation of the location. A disjunction is mapped into as many rows as the number of variables and cross-terms in the expression. In each row, a value is assigned to the corresponding variable, all others are "don't care" values. The last element of each of these rows contains the value of the location as shown in Figure 8.

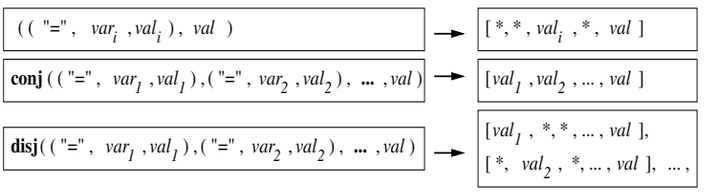


Fig. 8. Creating MDG tables rows from guarded updates

4.3 Algebraic Specification

We have to declare all sorts and functions before we use them in our MDG-HDL design descriptions. In the MDG tool, there is a default abstract sort *wordn* (for n-bit words) and a default concrete sort *bool* with the enumeration of [0,1]. Any other sorts must be explicitly declared. An ASM-IL representation preserves the enumeration for each variable. Based on this, we declare a concrete sort for each different enumeration. Since different abstract sorts can be distinguished in the ASM-SL model, we declare different abstract sorts for each one.

All functions and cross-terms are also declared in the algebraic specification in the same way. This includes uninterpreted functions and cross-terms. We declare any function that occurs in the ASM-IL expressions in the algebraic specification according to its arguments and domain sorts. These sorts are derived from the expressions where the function occurs. Figure 9 below illustrates the mapping function that we use to generate the MDG algebraic specifications.

4.4 Variable Order

MDGs have some restrictions on the order of abstract variables and cross-operators. In order to obey these restrictions, we explore all functions and cross-operators in the ASM-IL expressions and order the variables according to the dependencies between abstract variables themselves and also between abstract variables and cross-terms or functions. If a variable *var1* depends on another variable (or function) *var2*, then *var1* is sorted above *var2* in the order file. Also if a cross-term *f* depends on a variable *var1*, then *var1* should appear above *f*.

5 Case Study: Island Tunnel Controller

In this section, we provide a case study based on the Island Tunnel Controller (ITC) example [19] in order to illustrate the proposed ASM-MDG interface. The ITC is used to control two traffic lights for a tunnel that connects an island to the mainland as shown in Figure 10. There are four sensors to detect vehicles at both sides of the tunnel: *ie* at the entrance of the island (*carEntering* in ASM), *ix* at the exit of the island (*carExiting* in ASM), *me* at the entrance of the mainland and *mx* at the exit of the mainland. There are four output signals to

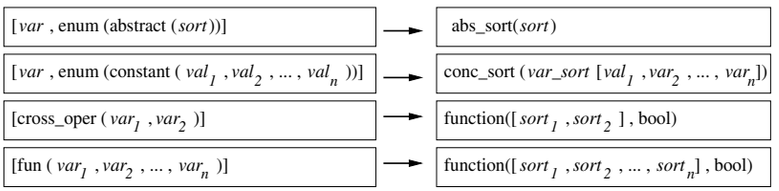


Fig. 9. Declarations of functions and sorts in the algebraic specifications

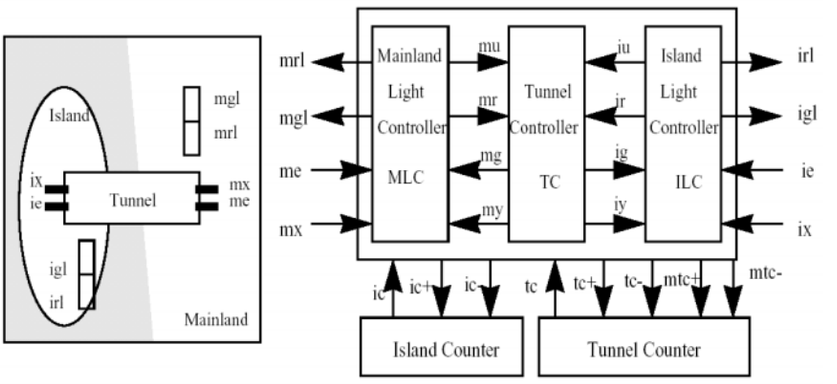


Fig. 10. Island Tunnel Controller

control the traffic lights at both sides: *irl* for island red light, *igl* for island green light, *mrl* for mainland red light and *mgl* for mainland green light. The ITC is specified using three communicating controllers and two counters as shown in Figure 10.

The tunnel allows a maximum number of cars to be in the island at one time. In ASM this number can be taken as a parameter of an abstract type that represents any natural number. We can then define a cross-operator for the operation "$islandCounter < n$". This allows the modeling of the controller for any number of cars. The advantage of modeling designs in ASM over MDG-HDL might not be clear in this hardware example since MDG-HDL is a hardware oriented language. However, from our experience it is easier and more convenient to model the controller behavior in ASM rather than to produce the MDG-HDL table representation, especially for large systems. Moreover, this example nicely illustrates the advantage of using abstract types that are supported by our framework. In contrast to other modeling and verification techniques, we are able to verify this system for any arbitrary counter size. We choose the Island Light Controller (ILC) as part of the design to develop two models in ASM: specification and implementation[2]. Figure 11 shows the state transition diagram for the ILC. We model this in ASM by defining a free type that represents the states of the ILC as follows:

```
freetype IS_SORT == {green, red, exiting, entering}
```

Increment and decrement operations on the counters are generally infinite mappings over integers. In our model, we specify those as abstract static func-

[2] The full specification models in ASM as well as the generated MDG-HDL models can be obtained at: `http://hvg.ece.concordia.ca/Tools/ASMMDG/ITC/`

tions, which maps between abstract values. These functions are left uninterpreted in our transformation.

External functions are used to represent environment operations for detecting vehicles at the entrance or exit in addition to signals from the Tunnel Controller, e.g. `external function carEntering: BOOL`

We describe the dynamic control of the controller states using the dynamic function: *islandState*.

```
dynamic function islandState: IS_SORT initially red
```

All Boolean outputs of this controller are also described by *dynamic functions*, e.g. `dynamic function itc_min: BOOL initially undef` which declares the island tunnel counter. We then describe the behavior of the system using *if-then-else* rules. One example is shown below for the *entering* state. Note that the value of *carEntering* is indicated by the sensor *ie* (see Figures 10 and 11).

The implementation of the ILC is shown in Figure 12. We use *static functions* to define primitive gates (AND, OR, NAND, etc.). An example is shown below for an OR gate with two inputs.

```
static function or2 (in0,in1) ==
if in0 = true  or  in1 = true
then true
else false
endif
```

We describe the behavior of output signals in addition to internal signals in one transition ITC_IMPL. The time delay between signals is modeled using a transition DFF that represents a flip flop component.

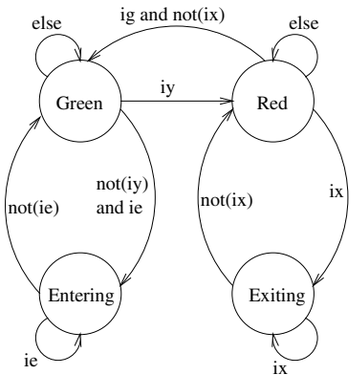


Fig. 11. State Transition for the ILC

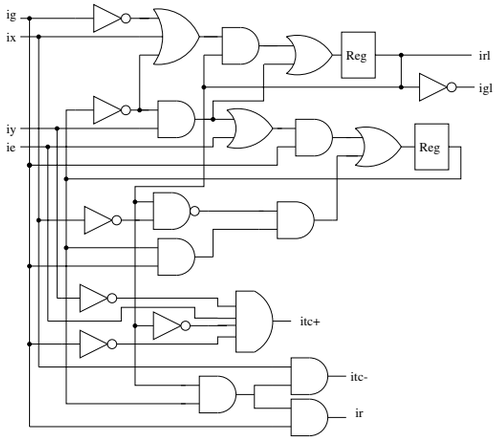


Fig. 12. ILC Implementation

```
transition DFF ==
block
x := Dx
y := Dy
z := Dz
endblock
```

We applied our interface tool on both of these models, generating the ASM-IL models for each one, then we built our target syntax MDG-HDL structural and behavioral models in addition to the algebraic specification and the variable order. The algebraic specification contain the declaration of all abstract and concrete sorts used in the modeling, e.g., the declaration of the concrete sort is generated as:

```
conc_sort(islandState_type,[entering_,exiting_,green_,red_]).
```

Once the required files generated, the application of the formal verification procedures of the MDG tool (e.g., equivalence checking or model checking) on the design models is straightforward. This is done like any MDG tool verification session, which compiles the MDG-HDL model files, runs the corresponding verification procedure, and either reports success or failure. In the latter case a counter-example is generated which contains a trace leading from the initial state to the state where the checking failed.

6 Conclusions

In this paper, we introduced a new interface from the ASM Workbench to the MDG tool. This enables the user of ASM to exploit the fully automated verification techniques that are provided by the MDG tool, namely equivalence checking

and model checking. In contrast to other model checkers, the MDG tool provides a very simple means for abstracting an ASM model from infinite domains in order to render it feasible for the automated checking approaches. The interface has been implemented and tested on a case study. It automatically transforms ASM-SL models into MDG-HDL code, which is the input language of the MDG tool. The first step ASM to ASM-IL is tailored to the ASM Workbench, however, if any other ASM tool would provide the same step into some kind of ASM-IL, we certainly could reuse our mapping into MDG.

The approach is built upon existing work that transforms ASM to an intermediate language. Since no structure from the original ASM model is preserved in this step, structural information needed to be regained in the second step that transforms the intermediate language to the different parts of the MDG-HDL code. Apart from this code, the interface produces a static variable ordering, necessary for the checking procedures, that satisfies the restrictions given by the MDG approach.

Although the case study, the Island Tunnel Controller, is a hardware example and could have also been modeled in MDG-HDL, the benefits of extending the MDG tool with a general high-level modeling language like ASM are easy to realize once the user focuses on non-hardware problems. Furthermore, the case study nicely demonstrates the benefits of the MDG tool over ordinary ROBDD-based tools, like SMV: Parameterized models can be checked without instantiating the parameters. In the case of the Island Tunnel Controller, the model could be checked for an arbitrary number of allowed cars in the tunnel.

References

[1] S. Balakrishnan A Hierarchical Approach to the Formal Verification of Embedded Systems Using MDGs. Master's Thesis, Concordia University, Department of Electrical and Computer Engineering, November 1999. 283

[2] D. Beauquier and A. Slissenko. The Railroad Crossing Problem: Towards Semantics of Timed Algorithms and their Model-Checking in High-Level Languages. In M. Bidoit and M. Dauchet (eds.), Proceedings of TAPSOFT'97, LNCS 1214, Springer-Verlag, 1997. 279

[3] E. Cerny, F. Corella, M. Langevin, X. Song, S. Tahar, Z. Zhou. Automated Verification with Abstract State Machines Using Multiway Decision Graphs. In T. Kropf (ed.), Formal Hardware Verification: Methods and Systems in Comparison, LNCS 1287, State-of-the-Art Survey, Springer-Verlag, 1997, pp. 79-113. 278, 282, 283

[4] F. Corella, Z. Zhou, X. Song, M. Langevin and E. Cerny. Multiway Decision Graphs for Automated Hardware Verification. Formal Methods in System Design, Vol. 10, February 1997, pp. 7-46. 280, 282

[5] G. Del Castillo. The ASM Workbench: A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models. Ph.D. Thesis, Heinz Nixdorf Institute, Paderborn, Germany, 2000. 279, 280

[6] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-level Specification Language. In S. Graf and M. Schwartzbach (eds.), Tools and Algorithms for the Construction and Analysis of Systems. LNCS 1785, Springer-Verlag, pp. 331- 346, 2000. 279, 280

[7] Y. Feng, E. Cerny. Term Ordering Problem on MDG. In Proc. ACM 12th Great Lakes Symposium on VLSI, New York City, New York, USA, April 2002, pp. 160-165. 283

[8] J. K. Huggins. Abstract State Machines home page. EECS Department, University of Michigan. `http://www.eecs.umich.edu/gasm/`. 278, 279

[9] S. Katz and O. Grumberg. A Framework for Translating Models and Specification. In K. Sere and M. Butler (eds.), Integrated Formal Methods. LNCS 2335, Springer- Verlag, 2002. 279

[10] S. Kort, S. Tahar, and P. Curzon. Hierarchical Verification Using an MDG-HOL Hybrid Tool. In: T. Margaria and T. Melham (eds.), Correct Hardware Design and Verification Methods, LNCS 2144, Springer Verlag, 2001, pp. 244-258. 279

[11] T. Kropf. Introduction to Formal Hardware Verification, Springer-Verlag, 1999. 278, 280

[12] Y. Mokhtari, M. Shirazipour, and S. Tahar. A Case Study on Model Checking and Refinement of Abstract State Machines. Proc. Eight International Conference on Computer Aided Systems Theory, Las Palmas de Gran Canaria, Canary Islands, Spain, February 2001, pp. 239 242.

[13] C. Plonka. Model checking for the design with abstract state machines. Master's thesis, University of Ulm, Department of Applied Information Processing, 2000. 279

[14] M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled (eds.), Computer Aided Verification, LNCS 1633, pages 431-442. Springer- Verlag, 1999. 279

[15] N. Shankar. Symbolic Analysis of Transition Systems. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele (eds.), Abstract State Machines, Theory and Applications, ASM 2000, LNCS 1912, pp. 287-302, Springer-Verlag, 2000. 279

[16] S. Tahar, X. Song, E. Cerny, Z. Zhou, M. Langevin and O. Ait-Mohamed. Modeling and Verification of the Fairisle ATM Switch Fabric using MDGs. IEEE Transactions on CAD of Integrated Circuits and Systems, Vol. 18, No. 7, July 1999, pp. 956-972. 283

[17] K. Winter. Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin 2001. 279, 280, 283

[18] Y. Xu, E. Cerny, X. Song, F. Corella, O. Mohamed. Model Checking for First-Order Temporal Logic using Multiway Decision Graphs. In Computer Aided Verification, LNCS 1427, Springer Verlag, 1998, pp. 219-231. 282

[19] Z. Zhou, X. Song, S. Tahar, E. Cerny, F. Corella, M. Langevin. Formal Verification of the Island Tunnel Controller using Multiway Decision Graphs. In Formal Methods in Computer-Aided Design, LNCS 1166, Springer Verlag, 1996, pp. 233-246. 283, 287

[20] Z. Zhou and N. Boulerice. MDG Tools (v1.0) User's Manual. Dept. of Information and Operation Research, University of Montreal, Canada, 1996. 279, 282

[21] M. H. Zobair. Modeling and Formal Verification of a Telecom System Block using MDGs. M. A.Sc. Thesis, Concordia University, Department of Electrical and Computer Engineering, Montreal, Canada, April 2001. 283

ASMs versus Natural Semantics: A Comparison with New Insights

Sabine Glesner

Institut für Programmstrukturen und Datenorganisation, Universität Karlsruhe
76128 Karlsruhe, Germany
http://www.info.uni-karlsruhe.de/~glesner

Abstract. We compare three specification frameworks for the operational semantics of programming languages, abstract state machines (ASMs) and the two incarnations of natural semantics, big-step and small-step semantics, with respect to two criteria: the range of imperative programming languages to which they are applicable and the way the program is used in the specifications and treated during the thereby defined execution. To reveal the fundamental differences between these three mechanisms, we investigate if there are automatic transformations between them. As a side effect, this leads to new insights concerning the classification of big-step and small-step semantics.

1 Introduction

Abstract state machines (ASMs) and natural semantics with its two incarnations big-step and small-step semantics are competing specification frameworks for the operational semantics of programming languages. In the ASM as well as in the natural semantics community, there exists an extensive engineering knowledge of how to use these specification mechanisms appropriately. This raises the widely debated question if there are fundamental differences between them. We compare ASMs and natural semantics according to two main criteria: First, we characterize them regarding the structure of imperative programming languages whose semantics can be defined with them. Secondly, we evaluate them with respect to the way the programs are treated in the specifications and during their thus defined execution. While both criteria are certainly coupled, it turns out that the second criterion really shows where the fundamental differences are.

To accomplish the desired comparison between ASMs and natural semantics, we define, if possible, automatic semantics-preserving transformations from one mechanism into the other. This proceeding is particularly helpful as it separates non-relevant discrepancies in notation from essential differences. Since semantics is defined operationally in these frameworks, each program is regarded as a state transition system. Hence, semantic equivalence means in our context that for each program, the state transitions during its execution are the same. If all specification mechanisms, ASMs as well as big-step and small-step semantics, could be applied for arbitrary programming languages, then we could hope to

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 293–309, 2003.
© Springer-Verlag Berlin Heidelberg 2003

find transformations in any desired direction. This is not the case and leads us directly to our classification of imperative programming languages. We distinguish between strictly compositional and non-strictly compositional programming languages. In a strictly compositional programming language, the semantics of each part of the program, which we regard in form of its abstract syntax tree, can be defined solely in terms of the semantics of its direct parts, i.e. subtrees. Big-step semantics defines semantics of programs recursively in terms of the semantics of their direct subtrees. This implies that big-step semantics can only define the semantics of strictly compositional programming languages. This does not hold for small-step semantics and ASMs.

Concerning the second criterion, treatment of programs, there are more similarities between big-step semantics and ASMs while small-step semantics is the outlier. Both big-step semantics and ASMs use the abstract syntax trees of programs in their specifications but do not modify it during program execution. In contrast, small-step semantics explicitly rewrites the abstract syntax trees during execution. In general, a small-step semantics defines a term-rewriting system. Starting with the original program as initial continuation program, during each state transition, the current continuation program is rewritten until the empty program is reached. In each state, the continuation program represents the computation which still needs to be done. In contrast, ASMs represent the remaining computation in a given state as a pointer to the node in the abstract syntax tree which is executed next. In each state transition, this pointer is updated.

We show that each small-step semantics can be transformed automatically into an equivalent ASM semantics and vice versa. We also prove that each big-step semantics can be transformed automatically into an equivalent ASM.

2 Semantics of Programming Languages

In general, the semantics of programs is compositional. Given a program in form of an abstract syntax tree, the semantics of each node can be defined directly given its immediate successors. Nevertheless, certain constructs in programming languages exhibit a semantics which is inherently not compositional. E.g., `goto`-statements may leave a program part and go to some other place which cannot be described via the predecessor or successor relation in abstract syntax trees. To define non-compositional semantics, we need *continuations*. A continuation tells us where to proceed with the computation. In compositional program constructs, the continuation specifies simply a child or the parent node. In general, the continuation denotes an arbitrary program node. Continuations can be computed already at compile time. Therefore, attributes are specified defining where to continue the computation. If the control flow branches at a node, then several such continuations are defined, each describing the succeeding computation depending on the branch direction. In this paper, we regard a program as an attributed abstract syntax tree (AST) whose attributes specify the continuations.

3 Defining Semantics with ASMs

Abstract state machines (ASMs) [Gur95, AW] have been used extensively in the definition of the semantics of programming languages, e.g. in the definition of C [GH93], of Java [SSB01] and of SDL [EGGP00]. Moreover, ASMs have been used successfully in proving the correctness of compilations, e.g. the correctness of the compilation of Prolog to the WAM [BR94], the correctness of the translation of Occam to transputer code [BD96] and in the Verifix project which deals with the construction of provably correct compilers [ZG97, GZ99]. During these projects, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. In this section, we summarize this particular use of ASMs. Our presentation generalizes the description in [GZ99].

Remark: Some ASM semantics (e.g. [SSB01]) modify the AST during program execution. We do not consider this here as it is not the typical case.

Abstract state machines (ASMs) are used to describe the semantics of programming languages operationally as state transition systems based on the abstract syntax trees. Part of the current state is the current task, a pointer to the node in the abstract syntax tree which is currently executed. During program execution, states are transformed into new states, thereby also updating the pointer to the current task. States are regarded as algebras over a given signature. During a state transition, the interpretation $\mathcal{I}$ of some of the function symbols may change. For example, if a function symbol S specifies the state of memory, then a variable assignment `x:=v` changes the interpretation $\mathcal{I}(S)$ of the function symbol S for argument `x`: $\mathcal{I}(S(\mathtt{x})) := \mathcal{I}(\mathtt{v})$ holds in the new state. Each n-ary function symbol is interpreted with an n-ary mapping. For each state transition, the interpretation of some function values might change. In general, an ASM consists of four components $(\Sigma \cup \Delta, \mathcal{A}, \mathit{Init}, \mathit{Trans})$: The signature is composed of two disjoint sorted signatures, the signature of the *static functions* Σ and the signature of the *dynamic functions* Δ. $\mathcal{A}$ is the *static algebra*, an order-sorted Σ-algebra interpreting the function symbols in Σ. *Init* is a set of equations over $\mathcal{A}$ which defines the *initial states* of $\mathcal{A}$. Finally, *Trans* is a set of *transition rules* for specifying the state transitions by defining or updating, resp., the interpretations of certain function values of functions in Δ. A $(\Sigma \cup \Delta)$-algebra is a state of the ASM iff its restriction to Σ is the static algebra $\mathcal{A}$. If q is a state, $f \in \Delta$ is a function symbol, and t_i are terms over $\Sigma \cup \Delta$ with interpretations x_i in q, then the update $f(t_1, \ldots, t_n) := t_0$ defines the new interpretation of f in the succeeding state q' as

$$q' \models f(x_1, \ldots, x_n) = \begin{cases} x_0 & \text{if for all } i,\ 0 \le i \le n,\ q \models t_i = x_i \\ f_q(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

A transition rule defines a set of updates which are executed in parallel:

if *Cond* **then** $\mathit{Update}_1 \ldots \mathit{Update}_n$ **fi**

If $q \models \mathit{Cond} = \mathit{true}$ in state q, then $\mathit{Update}_1 \ldots \mathit{Update}_n$ are executed in q.

When defining the semantics of programming languages, we use the abstract syntax tree as basis and attach meaning to it, cf. section 2. Thereby, we assume that the abstract syntax tree contains attributes defining all continuations, especially for the non-compositional changes of the control flow. The definition of the ASM models the program counter during program execution, thereby using the continuation attributes which might be split up according to the value of conditions (true case and false case). Here is the example of a transition rule defining the semantics of the while-loop, as stated in [GZ99]. CT (CT = current task) is the abstract program counter, $CT.TT$ (true task) is the true-continuation attribute of CT and $CT.FT$ (false task) is the false-continuation attribute of CT.

if $CT \in \mathit{While}$ **then**
  if $\mathit{value}(CT.\mathit{cond}) = \mathit{true}$ **then**
    $CT := CT.TT$
  else $CT := CT.FT$ **fi fi**

The semantics of each program node is described by a finite set of transition rules. Typically the condition of such a transition rule specifies the nodes in the abstract syntax tree (**While**-nodes in our example) for which the transition rule is applicable. The transition rules define updates, thereby employing child nodes (in our example $CT.\mathit{cond}$) as well as statically computed continuations ($CT.TT$ and $CT.FT$ in our above example). In the remainder of this paper, we assume that in an ASM definition which specifies the semantics of a programming language, each transition rule is of the following general form:

if $CT \in X$ **then**
  if $\mathit{applicability_conditions}$ **then**
    $CT := \mathit{new_CT}$; $\mathit{further_updates}$
  else $CT := \mathit{new_CT'}$; $\mathit{further_updates'}$ **fi fi**

4 Defining Semantics with Natural Semantics

Natural semantics [Plo81, Kah87] is a deductive method to define static and dynamic properties of programs. Thereby, axioms and inference rules specify semantic properties with respect to the abstract syntax. Natural semantics has been used extensively in the definition of programming languages. A prominent example is the complete specification of Standard-ML [MTH90]. Its revision [MTHM97] demonstrates that natural semantics specifications are very stable. Most of the modifications changed the semantics of ML itself rather than correcting errors in the original specification. Further examples for language specifications are the dynamic semantics of Eiffel [Att96], Eiffel// (Eiffel Parallel) [ACEL96], Esterel [Ber90], and in general imperative and object-oriented programming languages [GZ98, Gle99a, Gle99b]. Natural semantics has also been used successfully to prove properties of programs: The investigations in [DE99, Sym99, vON99] prove the static type safety of subsets of the Java programming language whereby specifications based on natural semantics are used.

There are two main variants of natural semantics, big-step semantics and small-step semantics (also called structural operational semantics (SOS)). While big-step semantics can only describe strictly compositional programming languages, small-step semantics is also able to handle non-compositional program constructs. We investigate both variants separately as they differ significantly wrt. our two criteria, i.e. their treatment of the abstract syntax trees and the range of programming languages to which they are applicable. In the following two subsections, we describe both of them, for more details consult e.g. [NN99].

Remark: The terminology is not consistent throughout the literature. Sometimes natural semantics refers only to big-step semantics, sometimes it comprises big-step as well as small-step semantics.

4.1 Big-Step Semantics

If the semantics of a programming language is strictly compositional, then we can define the semantics of an abstract syntax tree depending solely on the semantics of its direct subtrees. Consider e.g. the two inference rules for defining the semantics of the while-loop:

$$\frac{\mathsf{Eval}(\mathit{cond}, \sigma) = \mathit{false}}{< \textbf{while } \mathit{cond} \textbf{ do } S \textbf{ end}, \sigma > \ \rightarrow \ \sigma}$$

$$\frac{\mathsf{Eval}(\mathit{cond}, \sigma) = \mathit{true}, < S, \sigma > \ \rightarrow \ \sigma', < \textbf{while } \mathit{cond} \textbf{ do } S \textbf{ end}, \sigma' > \ \rightarrow \ \sigma''}{< \textbf{while } \mathit{cond} \textbf{ do } S \textbf{ end}, \sigma > \ \rightarrow \ \sigma''}$$

These two rules express that the body S of the loop is executed depending on the value of the condition *cond*. If it is executed, then the entire loop is executed recursively again. Typically, this kind of semantic description is only used for terminating computations. In this case, the second rule says that there exists a state transition from σ to σ'' if the condition *cond* evaluates to true, if the body S is executed by a state transition from σ to σ' and if there is a state transition from σ' to σ'' describing the recursive execution of the loop.

We can regard a natural semantics as a recursive procedure defined by inference rules. Each inference rule belongs to a production $X_0 ::= X_1 \cdots X_n$ of the abstract syntax. It has the following general form, whereby $X_{l_r} \in \{X_1, \ldots, X_n\}$, $1 \leq r \leq m$ and $X_{i_j} \in \{X_0, X_1, \ldots, X_n\}$, $1 \leq j \leq k$, m, k natural numbers:

$$\frac{\begin{array}{c}\mathsf{Eval}(X_{l_1}, \sigma_0) = \mathit{value}_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma_0) = \mathit{value}_m, \\ < X_{i_1}, \sigma_0 > \ \rightarrow \ \sigma_1, < X_{i_2}, \sigma_1 > \ \rightarrow \ \sigma_2, \cdots, < X_{i_k}, \sigma_{k-1} > \ \rightarrow \ \sigma_k\end{array}}{< X_0, \sigma_0 > \ \rightarrow \ \sigma_k}$$

The assumptions of an inference rule consist of two main parts, the evaluation conditions $\mathsf{Eval}(X_{l_r}, \sigma_0)$ and the "procedure calls" on direct successors of X_0. The evaluation conditions decide about the applicability of the inference rule in a given state σ_0. In the example of the while-loop, they express the value

$\mathsf{Eval}(cond, \sigma)$ of the condition *cond*. If the evaluation conditions are fulfilled, then the procedures, i.e. inference rules, for $X_{i_1}, \ldots, X_{i_k}$ are called in this order, each with the corresponding initial state $\sigma_0, \ldots, \sigma_{k-1}$ as input value and with the corresponding final state $\sigma_1, \ldots, \sigma_k$ as result. The X_{i_j}, $1 \leq j \leq k$, are the roots of direct subtrees of X_0. A particular X_{i_j} might be called several times with possibly different initial values or might not be called at all. Since we assume a strictly compositional programming language to be described with such inference rules, the semantics of the abstract syntax tree can be concluded solely from the semantics of its direct subtrees (in recursive cases also from its own semantics).

In natural semantics specifications, data structures define the values of the evaluation conditions and the states reached during program execution. These data structures are typically defined inductively by a term algebra over a fixed set of constructor functions. Additional (defined) functions are specified by equations defining recursively the effect of these functions on all constructor terms.

To summarize, big-step semantics describe state transitions for entire programs, i.e. abstract syntax trees. They formalize the execution of a program p as a transformation from an initial state σ into a final state σ', $< p, \sigma > \ \rightarrow \ \sigma'$, specified in the conclusion of the inference rule. This state transition is split up into a sequence $\sigma = \sigma_0 \rightarrow \sigma_1 \rightarrow \cdots \rightarrow \sigma_k = \sigma'$ of state transitions such that each individual state transition $\sigma_{i-1} \rightarrow \sigma_i$, $1 \leq i \leq k$, is described by exactly one of the assumptions. Since the subsequent program part to be executed next is specified implicitly by this linear ordering of the states, $\sigma_0 \rightarrow \sigma_1 \cdots \rightarrow \sigma_k$, we do not need to use any continuation attributes explicitly. This suffices because the described programming languages are assumed to be strictly compositional.

4.2 Small-Step Semantics

Small-step semantics, also called structural operational semantics (SOS), concentrate on individual steps of program execution and how these single steps are integrated in the overall execution. Assumptions of inference rules formalize smaller steps while their embedding into the larger program context is defined in the conclusion. Individual steps are described in the axioms. Such an individual step is either termination of execution $< p, \sigma > \ \rightarrow \ \sigma'$ in the final state σ' or it is a state transition $< p, \sigma > \ \rightarrow \ < p', \sigma' >$ denoting that the execution of p in state σ yields a new program p' to be executed in the succeeding state σ'. p' is often called *continuation*. In this paper, we call it *continuation program* to distinguish it from the statically computable continuation attributes described in section 2. In most cases, p' is a direct subtree of p or composed from direct subtrees of p. The conclusions of inference rules define the embedding of such program parts into their larger context. In the case of compositional semantics, this context is simply the parent node in the abstract syntax tree. In general, arbitrary continuations are possible, allowing for the description of non-compositional semantics. As typical examples for small-step definitions, consider these inference rules:

$$\frac{<S_1,\sigma> \rightarrow <S_1',\sigma'>}{<S_1;S_2,\sigma> \rightarrow <S_1';S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(cond)=true}{<\textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2,\sigma> \rightarrow <S_1,\sigma>}$$

$$\frac{<S_1,\sigma> \rightarrow \sigma'}{<S_1;S_2,\sigma> \rightarrow <S_2,\sigma'>} \qquad \frac{\mathsf{Eval}(cond)=false}{<\textbf{if } cond \textbf{ then } S_1 \textbf{ else } S_2,\sigma> \rightarrow <S_2,\sigma>}$$

$$< \textbf{skip}, \sigma > \ \rightarrow \ \sigma$$

$$< \textbf{while } cond \textbf{ do } S, \sigma > \ \rightarrow < \textbf{if } cond \textbf{ then } (S; \textbf{while } cond \textbf{ do } S) \textbf{ else skip}, \sigma >$$

The first two inference rules in the left column describe how the execution of a sequence of statements S_1 is integrated into a larger context, namely the sequence of statements $S_1; S_2$. The first two rules on the right-hand side specify the execution of the if-statement. The first axiom defines the effect of the skip-statement. Finally the last axiom describes the while-loop by reducing its semantics to the semantics of the if-statement. In contrast to a big-step semantics, these rules describe execution in a bottom-up style: execution of smaller program parts is integrated into the execution of larger parts of the program, without paying attention to the overall state transition performed by the entire program. The structure of the inference rules does not need to reflect the structure of the program. For example, the semantics of the while-statement is not defined in terms of its sub-statements but by a new program which has been built from the sub-statements. (Note that the if-statement in the last axiom is not part of the original program but created upon application of this axiom.) In general, also statically computed continuation attributes (cf. section 2) can be used. As an example, consider the semantics of the goto-statement:

$$< \textbf{goto } L; \sigma > \ \rightarrow \ < L.continuation, \sigma' >$$

This axiom defines a non-compositional semantics since the control-flow of the program branches to another arbitrary part of the program, denoted by the continuation of L, $L.continuation$.

The general form of a small-step inference rule is as follows: Let $X_0 ::= X_1 \cdots X_n$ be a production of the abstract syntax, $X_{l_r} \in \{X_1, \ldots, X_n\}$, $1 \leq r \leq m$, m a natural number, X_i' is an arbitrary program built from X_i or its direct subprograms, i.e. direct subtrees of its abstract syntax tree, $1 \leq i \leq n$, X_0' is an arbitrary program built from X_0, from its subprograms $X_1 \cdots X_n$, from X_i' and from the continuations of $X_0, \ldots, X_n$.

$$\frac{\mathsf{Eval}(X_{l_1}, \sigma) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = value_m, \quad < X_i, \sigma > \ \rightarrow \ < X_i', \sigma' >}{< X_0, \sigma > \ \rightarrow \ < X_0', \sigma' >}$$

In its evaluation part, $\mathsf{Eval}(X_{l_1}, \sigma) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = value_m$, the inference rule describes conditions for which the rule is applicable. The state transition $< X_i, \sigma > \rightarrow \ < X_i', \sigma' >$ defines the execution of X_i in state σ and gives us a new continuation program X_i' to be executed in state σ'. The conclusion of the inference rule specifies how this single transition $< X_i, \sigma > \ \rightarrow < X_i', \sigma' >$ can be integrated into the larger context X_0 whose execution in state σ yields the new continuation program X_0' to be executed in the new state σ'.

The data structures necessary to define the values of the evaluation conditions and the states reached during program execution are defined in the same way as in big-step semantics definitions, cf. subsection 4.1.

In a small-step semantics, the program to be executed is an explicit part of the state. Each state $< p, \sigma >$ contains a continuation program p. In the initial state, p is the original program while in the final state, p is simply the empty program. The axioms and inference rules of a small-step semantics define how to rewrite this program during each state transition.

5 Transformations between ASMs and Natural Semantics

Since big-step semantics can only define strictly compositional semantics, we cannot hope for an automatic transformation from ASMs or small-step semantics to big-step semantics. The reverse direction is possible. We prove that we can transform each big-step semantics into an equivalent ASM. Furthermore, we show that each ASM can be transformed automatically into an equivalent small-step semantics and vice versa. This implies that each big-step semantics can also be transformed into an equivalent small-step semantics.

5.1 Data Structures in the Specifications

ASMs as well as big-step and small-step semantics define state transitions by exploiting (more or less strictly) the structure of abstract syntax trees. Thereby data structures are defined to represent the states and values which are computed during program execution. In the ASM case, these data structures are defined by the signatures $\Sigma \cup \Delta$ of the static and dynamic functions, the set *Init* of equations defining the initial states, and implicitly by the transition rules which specify how to change their interpretation from one state to another. The signatures define a Herbrand universe. The set *Init* maps all terms into the same equivalence class which are equal under these equations. The transition rules define how to modify this Herbrand structure, i.e. the interpretation, from one state to the next. In natural semantics, the data structures are defined also as a term algebra based on *constructor functions*. Additional *defined functions* can be introduced by stating inductively how they operate on constructor terms. These data structures correspond directly to the states of an ASM and vice versa as they can be interpreted also by the same Herbrand structures.

5.2 From Big-Step Semantics to ASMs

A big-step semantics defines execution of programs top-down: the state transitions of an entire abstract syntax tree are composed from the state transitions of its direct subtrees and, in recursive definitions, also from its own state transitions. When transforming a big-step semantics into an ASM specification, we need to explicitly define the continuation attributes which are specified only implicitly by the top-down style of the big-step semantics. Therefore we define

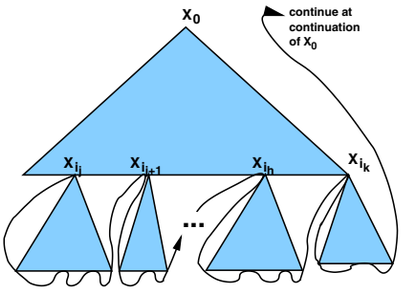


Fig. 1. Dynamic Continuations

a continuation attribute *cont* for each node in the abstract syntax tree. Since a node X_0 may be called recursively, these continuation attributes must also contain the continuations of all active calls of this node X_0. We organize the continuations in a stack (with the usual stack operations). We attach a dynamic stack attribute to each node in the abstract syntax tree. Its value during program execution is part of the current state.

It is important to observe that a big-step semantics defines individual state transitions only at the leaves of an abstract syntax tree. For all inner nodes, the inference rules specify how to compose the overall state transition sequence in the conclusion from the state transitions of the assumptions. When defining an equivalent ASM, the idea is to define rules modifying the state for the leaves of the abstract syntax tree. Thereby we use the function *update* taking two arguments σ and σ'. It maps the current ASM state σ to the new state σ' and can be defined easily (cf. remarks in subsection 5.1). Moreover, the rules for the inner nodes of the abstract syntax tree adjust the continuations. The idea is that the most right leaves (wrt. to the ordering of the nodes in the assumptions of the applied inference rules) of each subtree contain the continuation of the root of this subtree, cp. figure 1. We need to update the continuations sequentially from "right to left" wrt. the ordering of the assumptions. Since the ASM rules allow only for the specification of updates to be executed in parallel, we need to introduce several ASM rules per inference rule. The current task CT is a pair (X, n) where X is a pointer to the current node in the AST and n denotes the n-th update rule for X which needs to be executed next.

Definition 1. *Let* **Spec** *be a big-step semantics as defined in subsection 4.1, i.e. a set of axioms and inference rules. Then the corresponding ASM* $\mathbf{ASM_{Spec}}$ *is defined by the following transition rules:*

- *For each axiom* $< X, \sigma > \quad \rightarrow \quad \sigma'$*, the corresponding transition rule is defined:*

  if $CT \in (X, 0)$ **then**

  $update(\sigma, \sigma')$; $CT := (cont(X).top, 0)$; $cont(X) := cont(X).pop$ **fi**

– *For each inference rule of the general form*

$$\frac{\mathsf{Eval}(X_{l_1},\sigma_0) = value_1, \ldots, \mathsf{Eval}(X_{l_m},\sigma_0) = value_m,\quad < X_{i_1},\sigma_0 > \ \rightarrow\ \sigma_1, < X_{i_2},\sigma_1 > \ \rightarrow\ \sigma_2, \cdots, < X_{i_k},\sigma_{k-1} > \ \rightarrow\ \sigma_k}{< X_0,\sigma_0 > \ \rightarrow\ \sigma_k}$$

the corresponding transition rules are defined as:

if $CT \in (X_0, 0)$ **then**
  if $\mathsf{Eval}(X_{l_1}) = value_1$ **and** ... **and** $\mathsf{Eval}(X_{l_m}) = value_m$ **then**
    $cont(X_{i_k}) := cont(X_{i_k}).push(cont(X_0).top);\ cont(X_0) := cont(X_0).pop;$
    $CT := (X_0, k-1)$ **fi fi**
if $CT \in (X_0, k-1)$ **then**
  $cont(X_{i_{k-1}}) := cont(X_{i_{k-1}}).push(X_{i_k}); CT := (X_0, k-2)$ **fi**
...
if $CT \in (X_0, 2)$ **then**
  $cont(X_{i_1}) := cont(X_{i_1}).push(X_{i_2}); CT := (X_0.X_{i_1}, 0)$ **fi** ◇

To prove that the semantics of the ASM $\mathbf{ASM_{Spec}}$ defines the same semantics as the big-step semantics **Spec**, we need to show that for each program, the state transitions are the same in both specifications $\mathbf{ASM_{Spec}}$ and **Spec**.

Theorem 1. *Let* **Spec** *be a big-step semantics and* $\mathbf{ASM_{Spec}}$ *the corresponding ASM. The state transitions are the same for each program in both specifications.*

Proof. State transitions happen only at the leaves of the AST. The continuation of a node X is a reference to the node where the computation is to be continued after the computation of X is finished. The computation of a node and subtree X is finished when all its leaves are computed. Therefore the leaf processed at last has a continuation pointing to the continuation of X. Since a node might be called recursively, the different calls and their continuations are superimposed recursively in the abstract syntax tree. Since the continuations are organized in a stack, they represent the nested recursive structure properly. To prove that the state transitions of **Spec** and $\mathbf{ASM_{Spec}}$ are the same, we distinguish between terminating and non-terminating programs. For the terminating case, we do induction on the height of the abstract syntax tree X and its run-time expansion.

Base Case: X is a leaf described by axiom $< X, \sigma > \ \rightarrow\ \sigma'$. Clearly the state transition $update(\sigma, \sigma')$ gives us the same new state σ'. The deletion of the top continuation reference removes the current recursive frame.

Induction Step: For the computation of $X_0, X_{i_1}, \ldots, X_{i_k}$ need to be computed. We can assume that for $X_{i_1}, \ldots, X_{i_k}$, the state transitions are the same in the big-step semantics **Spec** and in $\mathbf{ASM_{Spec}}$ (induction hypotheses). It remains to show that the continuations are correct. Due to the updates $cont(X_{i_j}) := cont(X_{i_j}).push(X_{i_{j+1}})$ for $1 \leq j \leq k-1$, $X_{i_{j+1}}$ is computed directly after X_{i_j}, $1 \leq j \leq k-1$. The adjustment of the continuations from "right to left" makes sure that the stacking of the continuations is correct for the case that there exists $j \leq h, j, h \in \{1, \ldots, k\}$ such that $X_{i_j} = X_{i_h}$. When processing the subtree

marked with X_{i_J} (which equals X_{i_h}), first the continuation of the i_j-th subtree needs to be taken, then the continuation of the i_h-th subtree. Finally, after X_{i_k}, $cont(X_0)$ is executed so that computation either stops if X_0 is the root of the program or continues at the continuation of X_0. The deletion of the continuation of X_0, $cont(X_0).pop$, removes the current recursive frame at X_0. Since the continuation of X_0 is stored in the right most leaf of the subtree located at X_0, this continuation $cont(X_0)$ is not needed any more. Whenever this right-most leaf is reached, the execution will continue directly at $cont(X_0)$.

Induction Step for Non-terminating Programs: The above proof is only valid if the programs terminate. Only then the expansions of $X_{i_1}, \ldots, X_{i_k}$ are truly smaller than the expansion of X_0. If the program does not terminate, then there is one l, $1 \leq l \leq k$, such that $X_{i_1}, \ldots, X_{i_{l-1}}$ are truly smaller than X_0 and such that X_{i_l} is the first subtree with infinite height. The computation will get stuck in X_{i_l}. To prove that both specifications **Spec** and $\mathbf{ASM_{Spec}}$ show the same state transition behavior, we need to show that the execution at the root of X_{i_l} starts with the same state. This state is σ_{l-1}. The state at the root of X_{i_l} is the same for both specifications. Hence, we can conclude that both specifications enforce the same state transition behavior. If this were not the case, then there would be a smallest number of state transitions after which they are different. In the state before they would be the same. But then they must also be the same in the proceeding state. □

5.3 From Small-Step Semantics to ASMs

Small-step semantics define the execution of programs by recursively defining how to transform an initial state as well as a given program itself stepwise into a new state and a new program. This means that a small-step semantics does not define execution by a (eventually recursive) walk through the abstract syntax tree as it is the case in a big-step semantics. Rather the program is treated as a term which is rewritten during execution until it is reduced to the empty tree. This term to be reduced is also called *continuation*.

The axioms and inference rules in a specification define a recursive rewriting procedure. An axiom $< X, \sigma > \ \rightarrow \ \sigma'$ or $< X, \sigma > \ \rightarrow \ < X', \sigma' >$, resp., states that the current continuation X is to be replaced by the empty tree or the new program X', resp. An inference rule of the general form, cf. subsection 4.2, calls the rewriting procedure recursively on one of the direct subtrees X_i of the current program. After its completion, the rewriting procedure modifies its continuation and state by possibly integrating the results of the recursive call. The detailed recursive algorithm is stated in figure 2 in a pseudo-Pascal notation.

In general, this is not a pure term rewriting procedure since nodes may have static continuation attributes, cf. the goto-definition in subsection 4.2, which might point to arbitrary nodes in the original program tree. So whenever we talk about a subtree of the original program, we do not only mean the subtree itself but the transitive closure of all subtrees to which static continuations point.

proc $eval_AST(Cur_AST, state) : (New_AST, new_state)$;
  if $Cur_AST = nil$ **then**
    $(New_AST, new_state) := (nil, state)$;
  fi;
  if $Cur_AST \in X$ **and** $< X, \sigma > \ \rightarrow \ \sigma' \in Spec$ **then**
    $new_state := update(\sigma, \sigma')[\sigma/state]$;
    $New_AST := nil$;
  fi;
  if $Cur_AST \in X$ **and** $< X, \sigma > \ \rightarrow \ < X', \sigma' > \ \in Spec$ **then**
    $new_state := update(\sigma, \sigma')[\sigma/state]$;
    $New_AST := X'[X/Cur_AST]$;
  fi;
  if $Cur_AST \in X_0$
  and $\dfrac{\mathsf{Eval}(X_{l_1}, \sigma) = value_1, \ldots, \mathsf{Eval}(X_{l_m}, \sigma) = value_m, \quad < X_i, \sigma > \rightarrow \ < X_i', \sigma' >}{< X_0, \sigma > \ \rightarrow \ < X_0', \sigma' >} \in Spec$
  and $\exists$ *direct subtree* $X_i(Cur_AST)$ *of* Cur_AST *such that* $X_i(Cur_AST) \in X_i$
  and $Eval(X_{l_1}(Cur_AST, state)) = value_1$ **and** $\cdots$
  and $Eval(X_{l_m}(Cur_AST, state)) = value_m$ **then**
    $(Cur_AST', state') := eval_AST(X_i(Cur_AST), state)$;
    $(New_AST, new_state) := eval_AST(X_0'[X_0/Cur_AST, X_i'/Cur_AST'], state')$;
  fi;
  return (New_AST, new_state);
end_proc

(AST stands for *abstract syntax tree*.)

Fig. 2. Meaning of a Small-Step Semantics

The algorithm in figure 2 can easily be transformed into an ASM definition. Therefore the recursion is eliminated by transforming the recursive procedure into a while-loop which runs until the program is reduced to the empty tree. In the usual way, the nested recursive calls at run-time are modelled by a stack whose entries are tuples of the current continuation (i.e. program) and the current state, carrying the state of computation of the individual recursive calls. This is a trivial standard proceeding to eliminate recursion. The resulting while-loop can easily be restated as an ASM: The while-loop still contains the four if-statements as the original recursive procedure. It is straightforward to transform these if-statements into four corresponding ASM transition rules. The content of the stack of the while-loop during execution becomes the state of the ASM.

This resulting ASM is different from the ASMs typically defined when specifying the semantics of programming languages, cf. section 3. It does not define how to traverse the abstract syntax tree, e.g. by using a current task CT. Even though in many practical cases, given a small-step semantics, a human might easily be able to define walks through abstract syntax trees and a corresponding ASM, in general we cannot hope to find such an automatic transformation. This

is because a small-step semantics has many degrees of freedom in transforming a given program. It might duplicate subtrees, move subtrees from one part of the program to others by using the static continuation attributes, etc. This demonstrates the difference between small-step semantics on one hand and big-step semantics and ASM semantics on the other hand: While in big-step and (most) ASM semantics, the abstract syntax tree is a constant during program execution, it is modified during the execution defined by a small-step semantics.

5.4 From ASMs to Small-Step Semantics

An ASM specification defines the semantics of a programming language operationally based on the abstract syntax trees of the programs. The state of the ASM contains a reference CT to some node in the abstract syntax tree, pointing to the current task to be executed. ASM semantics specifications are able to describe not strictly compositional semantics. Therefore we cannot expect to find a transformation from ASM semantics specifications to big-step semantics because big-step semantics can only define strictly compositional semantics. But we can define a transformation from ASM semantics to small-step semantics.

The idea is to take the abstract syntax tree as (constant) dynamic continuation. The current task CT becomes part of the state: If σ is a state of the ASM and CT the current task during some point of program execution, then (CT, σ) is the state at the same point of program execution wrt. to the corresponding small-step semantics. Formally, for each transition rule in an ASM semantics,

if $CT \in X$ **then**
  if *applicability_conditions* **then**
    $CT :=$ *new_CT*; *further_updates*
  else $CT :=$ *new_CT′*; *further_updates′* **fi fi**

the corresponding inference rules are defined as follows:

$$\frac{\mathit{applicability_conditions}(AST, (\sigma, CT))}{< AST, (\sigma, CT) > \ \rightarrow < AST, (\mathit{further_updates}(\sigma), \mathit{new_CT}) >}$$

$$\frac{\neg\mathit{applicability_conditions}(AST, (\sigma, CT))}{< AST, (\sigma, CT) > \ \rightarrow < AST, (\mathit{further_updates'}(\sigma), \mathit{new_CT'}) >}$$

Again, as in subsection 5.2 and 5.3,we assume that the data structures of the ASM specification can be transformed easily into corresponding data structures of a small-step semantics. To prove that the defined transformation is correct, we need to show that for each program, the state transitions are the same wrt. to both specifications. Since the inference rules are not recursive, i.e., there are no state transitions specified in their assumptions, each execution will directly undertake the state transition of the conclusion of some inference rule whose assumptions are valid. This is the same as saying that some transition rule whose conditions (which are equivalent to the assumptions of the matching inference rule) are fulfilled will be executed. Therefore it follows immediately that the original ASM specification and the defined small-step semantics are equivalent.

6 Related Work

The results of our investigations are in contrast to the common understanding (cf. [NN99] or any other textbook or lecture notes of your choice) that big-step semantics can only describe terminating programs while small-step semantics is also suited for the description of non-terminating computations. This view is not adequate as our investigations show. Rather, it is the common interpretation of big-step semantics which fits only for terminating computations. In the traditional interpretation of big-step semantics, the assumptions must be true (terminating) in order to infer the conclusion. In our view, we ask if a (terminating or non-terminating) state transition sequence is consistent with the rules of the big-step semantics, allowing us to deal with non-terminating computations as well. This seems to be the more appropriate view anyway. Otherwise one could not determine whether a program has a semantics at all because this would be equivalent to solve the halting problem.

There are no other works comparing ASMs and big-step and small-step semantics wrt. the criteria "applicability to imperative programming languages" and "treatment of the AST". In particular, no transformations between the three mechanisms have been proposed. Only [ACK+00] proposes a mechanism to generate action notation environments from montages descriptions.

7 Conclusions

The three specification frameworks ASMs and small-step and big-step semantics vary significantly wrt. our two criteria "applicability to imperative programming languages" and "treatment of the abstract syntax tree". While big-step semantics can only define strictly compositional programming languages, ASMs and small-step semantics can also specify non-strictly compositional program constructs. Furthermore, big-step and most ASM semantics do not modify the abstract syntax tree during computation, in contrast to small-step semantics which explicitly defines a term-rewriting system that rewrites the program during execution until the empty program is reached. These differences are reflected in the transformations between them. We have shown that each ASM semantics can be transformed into an equivalent small-step semantics and vice versa. Furthermore we have proved that each big-step semantics can be transformed into an equivalent ASM semantics while the reverse direction cannot be expected.

From a theoretical point of view, these transformations are interesting as they reveal the unexpressed interpretations of the specification frameworks. ASMs and small-step semantics follow the idea that a program defines a state transition system whose execution can be observed. In contrast, the usual interpretation of big-step semantics defines how to construct finite state transition sequences. Our transformations indicate that for each specification mechanism, both interpretations are possible. The traditional classification – big-step only for terminating programs, small-step also for non-terminating programs – is not a classification of the specification frameworks but rather of their usual interpretations.

These transformations are also interesting from a practical point of view. In the ASM as well as in the natural semantics community, a remarkable engineering knowledge has emerged concerning the way in which specifications should be written to be useful for the purpose of semantics specification and translation verification. Having the transformations between these mechanisms in mind, we can transfer this engineering knowledge from one community to another.

In practice, most small-step semantics do not have the intention to define a term-rewriting system but rather incorporate the idea of defining a current task as in the ASM semantics. Therefore it would be interesting to define a simplified small-step semantics which allows for recursive semantic definitions on the tree structure of the program but does not permit to rewrite it. This seems to be sufficient for the usual applications. Moreover, it should be investigated if the three specification mechanisms ASMs and small-step and big-step semantics deal differently with multi-threaded and parallel programming languages. Finally, it should be investigated if they behave differently when used in automated theorem proving. This is an important criterion in the formal reasoning on the semantics of programming languages and the correctness of translations.

Acknowledgment

The author thanks A. Dold, W. Goerigk, G. Goos, H. Langmaack, V. Vialard, W. Zimmermann for many discussions on this research topic. Thanks also to the anonymous referees and E. Börger for valuable comments.

References

[ACEL96] Isabelle Attali, Denis Caromel, Sidi O. Ehmety, and Sylvain Lippi. Semantic-Based Visualization for Parallel Object-Oriented Programming. In *Proc. OOPSLA'96 (Object-Oriented Programming: Systems, Languages, and Applications)*, 1996. ACM Press, Sigplan Notices, Vol. 31, No. 10. 296

[ACK+00] M. Anlauff, S. Chakraborty, P. W. Kutter, A. Pierantonio, L. Thiele. Generating an Action Notation Environment from Montages Descriptions. *STTT*, 2000. 306

[Att96] Isabelle Attali. A Natural Semantics for Eiffel Dynamic Binding. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 18(5), 1996. 296

[AW] ASM-Website. `http://www.eecs.umich.edu/gasm/`. 295

[BD96] Egon Börger and Igor Durdanovic. Correctness of compiling Occam to Transputer Code. *Computer Journal*, 39(1):52–92, 1996. 295

[Ber90] Yves Bertot. Implementation of an Interpreter for a Parallel Language in Centaur. In *Proceedings of the 3rd European Symposium on Programming*, Copenhagen, Denmark, May 1990. Springer Verlag, LNCS 432. 296

[BR94] Egon Börger and Dean Rosenzweig. The WAM - definition and compiler correctness. In *Logic Programming: Formal Methods and Practical Applications*. North-Holland Series in Computer Science and Art. Int., 1994. 295

[DE99] Sophia Drossopoulou and Susan Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 41 ff. Springer Verlag, LNCS 1523, 1999. 296

[EGGP00] Robert Eschbach, Uwe Glässer, Reinhard Gotzhein, and Andreas Prinz. On the Formal Semantics of Design Languages: A compilation approach using Abstract State Machines. In *Abstract State Machines – ASM 2000, International Workshop on Abstract State Machines*, 2000. Local Proceedings, TIK-report 87, Swiss Federal Institute of Technology (ETH) Zurich. 295

[GH93] Yuri Gurevich and James K. Huggins. The semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, pages 274–308. Springer Verlag, LNCS 702, 1993. 295

[Gle99a] Sabine Glesner. Natural Semantics for Imperative and Object-Oriented Programming Languages. In *Informatik '99 - Informatik überwindet Grenzen, Proceedings der 29. Jahrestagung der Gesellschaft für Informatik*, 1999. GI-Gesellschaft für Informatik e.V., Springer Verlag. 296

[Gle99b] Sabine Glesner. *Natürliche Semantik für imperative und objektorientierte Programmiersprachen*. PhD thesis, Universität Karlsruhe, Fakultät für Informatik, February 1999. Shaker Verlag, Aachen, ISBN: 3-8265-6388-3. 296

[Gur95] Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995. 295

[GZ98] Sabine Glesner and Wolf Zimmermann. Using Many-Sorted Natural Semantics to Specify and Generate Semantic Analysis. In *Proc. Systems Implementation Conference (SI2000)*, 1998. IFIP WG 2.4, Chapman & Hall. 296

[GZ99] Gerhard Goos and Wolf Zimmermann. Verification of Compilers. In *Correct System Design*. Springer-Verlag, LNCS 1710, 1999. 295, 296

[Kah87] Gilles Kahn. Natural Semantics. In *Proc. 4th Annual Symp. on Theoretical Aspects of Computer Science (STACS'87)*, 1987. Springer, LNCS 247. 296

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990. 296

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997. 296

[NN99] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications, A Formal Introduction*. Published in 1992 by John Wiley & Sons, revised edition available at `http://www.daimi.au.dk/~hrn, 1999`. 297, 306

[Plo81] Gordon D. Plotkin. A structural approach to operational semantics. Report DAIMI FN-19, Comp. Sc. Dep., Aarhus University, Denmark, 1981. 296

[SSB01] Robert Stärk, Joachim Schmid, and Egon Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer Verlag, 2001. 295

[Sym99] Don Syme. Proving Java Type Soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, Springer Verlag, LNCS 1523, 1999. 296

[vON99] David von Oheimb and Tobias Nipkov. Machine-Checking the Java Specification: Proving Type-Safety. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, page 119 ff. Springer Verlag, LNCS 1523, 1999. 296

[ZG97] Wolf Zimmermann and Thilo Gaul. On the Construction of Correct Compiler Backends: An ASM Approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997. 295

Quantum Computing and Abstract State Machines

Erich Grädel and Antje Nowack

Mathematische Grundlagen der Informatik, RWTH Aachen
D-52056 Aachen, Germany
{graedel,nowack}@informatik.rwth-aachen.de

Abstract. The ASM thesis says that every algorithm, of any kind, can be modeled step by step and on its natural abstraction level by an abstract state machine. The thesis has been derived from basic principles for sequential algorithms, and for parallel synchronous algorithms. The main result of this paper is that the ASM thesis also holds for quantum algorithms.
We first show that, and how, a general model of quantum algorithms (based on the combination of a classical computer with quantum circuits) can be modeled by abstract state machines. Following the approach of Blass and Gurevich to parallel algorithms, we then formulate general postulates for quantum algorithms and show that every algorithm satisfying these postulates is simulated by an appropriate ASM. These "quantum ASMs" are special cases of the parallel ASMs of Blass and Gurevich, but the only unbounded parallelism is inside the quantum operations (unitary transformations and measurements).

1 Introduction

Abstract state machines (ASMs) provide a model for computation on abstract structures, which is the basis of a successful methodology for specification of hardware and software systems, with practical applications in industrial environments (see e.g. [4]). But ASMs are also interesting for the theory of computation, for instance for the very fundamental question of what constitutes an algorithm. The ASM thesis says that *"every algorithm of any kind is modeled, step by step and on its natural abstraction level, by an abstract state machine"*. There is considerable empirical evidence for this thesis, and in important cases, the thesis can actually be derived from basic postulates. In [3], Gurevich showed that this is the case for *sequential algorithms*. He formulated three basic postulates for sequential algorithms and proved that every algorithm satisfying these three postulates can be simulated, step by step, by a sequential ASM. It should be stressed that this notion of simulation or, in other words, the equivalence of two algorithms, is meant in a much more precise way than, for instance, in the Church-Turing thesis where simulation just means to have the same input-output-behaviour (to compute the same function). Here one requires complete behavioural equivalence: the two algorithms have the same states and initial

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 309–323, 2003.
© Springer-Verlag Berlin Heidelberg 2003

states, the same one-step transformation, and produce at every step the same output.

Later, Blass and Gurevich [1] extended that work to *parallel algorithms* which still work in sequential time, i.e. in discrete time steps governed by a global clock but with unbounded parallelism in each time step. For the more general model of distributed, asynchronous algorithms, there is so far no such treatment, but work seems to be under way.

However, it has been repeatedly asked, and so far not been answered in a satisfactory way, whether the ASM thesis also holds for nonstandard models of algorithms such as *quantum algorithms*. One of the reasons, why this question has not really been looked into, may be the widespread view in academia and industry that quantum computing provides a purely theoretical playground for mathematicians and physicists rather than "real algorithms". Indeed the main theoretical success of quantum computing is the polynomial-time quantum factorization algorithm by Shor [7], but on the practical side there are enormous difficulties for building quantum computers with more than a few qubits. The recent news that scientists at IBM have sucessfully factored the number 15 using a quantum computer with seven qubits [8] has probably not shaken that view. Still, the question remains, and provides a challenge for the advocates of the ASM thesis. First of all, to our present knowledge, the laws of physics seem not to exclude that large quantum computers can eventually be built, so the present state of quantum computers may just reflect our technological maturity rather than the potential of quantum computing. Second, quantum computing defines a mathematical model of computation (in fact: several models), and a thesis that comes with the ambition of sheding light on the general notion of algorithm should take care of this model as well.

Thus, we felt that the ASM thesis should apply also to quantum algorithms, and our intuition told us that it does. The purpose of this note is to explain that and how ASMs can model quantum algorithms.

2 Abstract State Machines

This paper assumes familiarity with the ASM model, in particular with Gurevich's paper [3] on sequential ASMs and the Blass-Gurevich paper [1] on parallel ASMs. Please consult these papers for unexplained definitions and terminology used here. In [1, 3] the ASM thesis for sequential and for parallel algorithms is derived from some basic postulates. Let us very briefly recall some of them. For sequential algorithms we follow here the presentation of [1] (rather than [3]) which makes the output that the algorithm produces at each step explicit, in form of a function out_A.

Postulates for Sequential Algorithms. A sequential algorithm with output is an object A satisfying the following three postulates.

S1: Sequential Time. A is associated with a set $\mathcal{S}(A)$ of *states*, a subset $\mathcal{I}(A) \subseteq \mathcal{S}(A)$ of *initial states*, a function $\tau_A : \mathcal{S}(A) \to \mathcal{S}(A)$, called the

one-step transformation of A, and a function out_A assigning to each state $X \in \mathcal{S}(A)$ a set (or multiset) of elements of X, called the output of A at state X.

S2: Abstract State. All states of A are first-order structures with the same finite vocabulary and the same universe. Both $\mathcal{S}(A)$ and $\mathcal{I}(A)$ are closed under isomorphisms, and any isomorphism between two states X and Y is also an isomorphism between $\tau_A(X)$ and $\tau_A(Y)$, and maps $out_A(X)$ to $out_A(Y)$.

S3: Uniformly Bounded Exploration. There is a finite set T of terms in the vocabulary of A such that, whenever two states X and Y coincide on T, then the update sets and outputs of A on X and Y are the same: $\Delta(A, X) = \Delta(A, Y)$ and $out_A(X) = out_A(Y)$.

We recall that the update set $\Delta(A, X)$ of A on state X represents the atomic changes in X produced by the algorithm A. It can be defined as the set of expressions $f(\overline{a}) = b$ that are true in $\tau_A(X)$ but not in X (where f is a function symbol, and $\overline{a}$ and b are elements of the universe).

A sequential ASM is an ASM that uses only

- function updates: $f(t_1, \ldots, t_m) := t_0$,
- parallel combination of two rules: `do` R_1, R_2 `enddo`,
- if-then-else rules: `if` φ `then` R_1 `else` R_2 `endif` where φ is a Boolean term (i.e. a formula without quantifiers), and
- output rules: `output` t, where t is any term.

The main result of [3] is that every sequential algorithm satisfying postulates (S1) - (S3), can be simulated, step by step, by a sequential ASM. Step by step simulation means that the two algorithms have the same states and initial states, the same one-step transformation, and the same output function.

Parallel Algorithms and Parallel ASMs. The notion of a parallel algorithm is meant here to capture algorithms that do still work in discrete time steps, governed by a global clock (as opposed to distributed, asynchronous algorithms), but with unbounded parallelism in each time step. Hence, parallel algorithms need not satisfy the Uniformly Bounded Exploration Postulate (S3). We will see that quantum algorithms are, in this sense, parallel algorithms.

The postulates for parallel algorithms cannot be explained here in detail. We limit ourselves to the aspects that we need for dealing with quantum algorithms. A parallel algorithm splits into smaller processes. Ultimately this splitting leads to atomic processes (called *proclets*), each of which executes the same (sequential!) algorithm. In the simplest case, the global update set of the algorithm is just the union of the 'local' updates produced independently by the proclets. In general however, parallel algorithms need to combine and integrate the updates produced by their proclets to a global update set in a more complicated way that involves communication between proclets.

Blass and Gurevich argue that this communication of proclets and the manipulation of messages and update sets is best described in terms of *multisets*

and multiset operations. The difference of multisets from sets is that multisets may have multiple occurrences of the same element. Formally, a multiset M over A is given by a function $\mathrm{Mult}_M : A \to \mathbb{N}$ giving for each $a \in A$ the multiplicity of $a \in M$ (only finite multiplicities are used here). We use the notation $\{\!\{\ldots\}\!\}$ to denote multisets. In particular, for a term $t(x)$, a set or multiset r, and a formula $\varphi(x)$, we write $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$ to denote the multiset M of all values of $t(x)$ subject to the condition that x is in r and satifies φ; more formally $\mathrm{Mult}_M(a) = \sum_{x:\varphi(x) \text{ and } t(x)=a} \mathrm{Mult}_r(x)$.

The **Background Postulate** for parallel algorithms states that the abstract states of parallel algorithms contain (among other things) all finite multisets of elements of the state and some basic operations on them. This is reflected in the definition of parallel ASMs which extend sequential ASMs in two essential ways:

Comprehension Terms: Given terms $t(x), r$, and $\varphi(x)$ one may build the comprehension terms $\{\!\{t(x) : x \in r : \varphi(x)\}\!\}$.

Unbounded Parallelism: If x is a variable, $R(x)$ is a rule and r is a term without free occurrences of x, then `forall` $x \in r$ `do` $R(x)$ `enddo` is a rule.

Note that both operations require parallelism (they do not satisfy the Uniformly Bounded Exploration Postulate). Also for parallel algorithms, the ASM thesis can be derived from appropriate postulates [1].

3 Models of Quantum Computation

For background on quantum computing we refer to the textbooks [2, 5, 6]. There are several theoretical models of quantum algorithms, including quantum circuits, quantum Turing machines, quantum automata, and quantum programming. What all these models have in common is that a *state* of a quantum algorithm is a *superposition of states* of a corresponding classical model.

Mathematically, a state of a quantum algorithm is a unit-length vector $|\psi\rangle \in H$ of a Hilbert space[1].

Quantum Bits and Quantum Registers. A quantum bit (shortly, qubit) is a vector in the two-dimensional Hilbert-space H_2 spanned by the orthonormal vectors $|0\rangle$ and $|1\rangle$. Hence, a qubit has the form $\alpha|0\rangle + \beta|1\rangle$ with $\|\alpha\|^2 + \|\beta\|^2 = 1$. If we *measure* a such a qubit, the probability to observe 0 is $\|\alpha\|^2$ and the probability to observe 1 is $\|\beta\|^2$. In the case of an n-qubit quantum register, we work in the 2^n-dimensional Hilbert-space $H_{2^n} = H_2 \otimes \cdots \otimes H_2$ with orthonormal basis $\{|w\rangle : w \in \{0,1\}^n\}$ which we always assume to be ordered lexicographically, with $|00\cdots0\rangle$ as first bases vector and $|11\cdots1\rangle$ as the last one (the order

[1] A Hilbert space is a complex vector space H, equipped with an inner product mapping pairs of vectors $|x\rangle, |y\rangle$ to complex numbers $\langle x|y\rangle$. Further H must be complete with respect to the norm $\|x\| := \sqrt{\langle x|x\rangle}$ induced by the inner product (i.e. every Cauchy sequence in H has a limit in H).

of a basis is important for the representation of linear transformations by matrices). A state of such a quantum register has the form $\sum_{w\in\{0,1\}^n} \alpha_w|w\rangle$ with $\sum_{w\in\{0,1\}^n} \|\alpha_w\|^2 = 1$.

A state $|\psi\rangle$ in the product space $H \otimes H'$ is decomposable if it can be written as a tensor product $|\psi\rangle = |\varphi\rangle \otimes |\varphi'\rangle$ with $|\varphi\rangle \in H$ and $|\varphi'\rangle \in H'$. Otherwise it is called *entangled*.

Quantum Circuits. The most influential model for quantum computation (probably due to the fact that most of the work on quantum algorithms is done by physicists) has been the quantum circuits model. There is not a uniform terminology here. While some authors use the notions quantum circuits, quantum algorithm and quantum computers interchangeably, others reserve the term quantum circuit for a sequence of quantum gates (with or without measurement), and view quantum computers or quantum algorithms as a more general model that may combine classical parts with quantum circuits and measurement steps. We follow here the second approach.

Definition 1. A *quantum gate on m qubits* is a unitary transformation U on a 2^m-dimensional Hilbert space H_{2^m}. Such a tranformation can be described by a unitary matrix[2] which we also denote by U. For our purposes, it is convenient to view this matrix as a function $U : \{0,1\}^m \times \{0,1\}^m \to \mathbb{C}$. The entry $U(x,y)$ gives the probability amplitude of the transition from base state $|y\rangle$ to base state $|x\rangle$.

Example. The Hadamard gate operates on H_2 and is defined by the matrix

$$H = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

which maps $|0\rangle$ to $|0'\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|1\rangle$ to $|1'\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Example. The CNOT (controlled NOT) gate is another gate which is used rather often. It operates on $H_4 = H_2 \otimes H_2$ and can be described by the operation $|a,b\rangle \to |a, a \oplus b\rangle$ or, equivalently, by the matrix

$$C = \begin{pmatrix} 1\,0\,0\,0 \\ 0\,1\,0\,0 \\ 0\,0\,0\,1 \\ 0\,0\,1\,0 \end{pmatrix}.$$

Definition 2. Let Ω be a fixed collection of quantum gates. A *quantum circuit* on n-qubits over Ω is a unitary transformation on H_{2^n} which is defined by a finite sequence of operations of the form

$$\text{“apply } U \text{ to the qubits } i_1, \ldots, i_m\text{”} \tag{1}$$

where U is an m-qubit gate from Ω, and $i_1, \ldots, i_m$ are distinct indices in $\{1, \ldots, n\}$.

[2] A matrix U is unitary, if its transpose conjugate U^* is its inverse: $U^*U = I$.

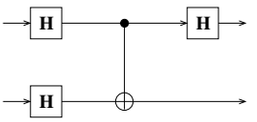
Mathematically, operation 1 is described by the matrix

$$U_{i_1 \dots i_m} := P^{-1}_{i_1 \dots i_m} (U \otimes I_{2^{n-m}}) P_{i_1 \dots i_m}$$

where $P_{i_1 \dots i_m}$ is a permutation matrix moving the qubits $i_1, \dots, i_m$ to the positions $1, \dots, m$ and $(U \otimes I_{2^{n-m}})$ is the tensor product of U (operating on the first m qubits) with the identity matrix of dimension 2^{n-m} (operating on the remaining qubits). Given the $2^m \times 2^m$-matrix U, and the numbers $i_1, \dots, i_m$ the entries of the $2^n \times 2^n$-matrix $U_{i_1 \dots i_m}$ are

$$U_{i_1 \dots i_m}(x, y) := \begin{cases} U(x_{i_1} \cdots x_{i_m}, y_{i_1} \cdots y_{i_m}) & \text{if } x_k = y_k \text{ for all } k \notin \{i_1, \dots, i_m\} \\ 0 & \text{otherwise.} \end{cases}$$

Remark. A popular notation for quantum circuits represents each qubit by a wire and the gates operating on them by boxes or circles (with special symbols for popular gates like Hadamard or CNOT). Here is an example.



The meaning of this picture is that before the application of the CNOT gate Hadamard gates are applied to both qubits separately. Afterwards, a Hadamard gate is applied to the first qubit. Mathematically, the circuit is described by the matrix $(H \otimes I_2)C(H \otimes H)$. Consider its operation on $|01\rangle$. The application of the Hadamard-gates leads to $\frac{1}{2}(|00\rangle + |10\rangle - |01\rangle - |11\rangle)$. The application of the CNOT-gate then leads to $\frac{1}{2}(|00\rangle + |11\rangle - |01\rangle - |10\rangle)$. The application of the Hadamard-gate to the first qubit finally leads to $\frac{1}{\sqrt{2}}(|10\rangle - |11\rangle)$.

Measurement. Let $H = M_1 + \cdots + M_k$ be a decomposition of the Hilbert space H into orthogonal subspaces. Then every $|\psi\rangle \in H$ can be written in a unique way as $|\psi\rangle = |\psi_1\rangle + \cdots + |\psi_k\rangle$, with $|\psi_i\rangle \in M_i$. Measurement of a state $|\psi\rangle$ (of unit length) with respect to the *observable* $\{M_1, \dots, M_k\}$ means that $|\psi\rangle$ is mapped to $\beta|\psi_i\rangle$ for one i, where i is picked with probability $\langle\psi_i|\psi_i\rangle$ and $\beta|\psi_i\rangle$ has unit length.

While one can in principle do measurements with respect to any orthogonal decomposition of the state space, in practice it normally suffices to do measurements with respect to one or more qubits in the computational basis. More precisely, we need operations

"measure with respect to qubits $i_1, \dots, i_m$"

which means that the current state $|\psi\rangle = \sum_{x \in \{0,1\}^n} \alpha(x)|x\rangle$ is written as $|\psi\rangle = \sum_{u \in \{0,1\}^m} |\psi_u\rangle$ where $|\psi_u\rangle = \sum_{x : x_{i_1} \cdots x_{i_m} = u_1 \cdots u_m} \alpha(x)|x\rangle$. We pick one u according to the probability distribution $p(u) = \sum_{x : x_{i_1} \cdots x_{i_m} = u_1 \cdots u_m} \|\alpha(x)\|^2$, and

project $|\psi\rangle$ to $\beta|\psi_u\rangle$ with $\beta = 1/\sqrt{p(u)}$ so that the resulting vector has again unit length.

We view a quantum algorithm as a combination of a classical sequential algorithm with quantum circuits. In its classical phases the algorithm may prepare any computational basis state $|x\rangle$ (where $x \in \{0,1\}^n$ for arbitrary n) and in its quantum phases a quantum circuit is applied. To summarize, a general model of quantum algorithms can be described as follows.

Classical and Quantum Part. A quantum algorithm is a combination of a classical algorithm with quantum circuits. The collection Ω of quantum gates that are used by the algorithm is fixed, but the quantum circuits used by the algorithm may work with any number n of qubits.

Quantum Mode. In quantum mode, a quantum circuit on some number n of qubits is applied to a computational basis state $|x\rangle = |x_0 \dots x_{n-1}\rangle$. As result the circuit produces a (usually entangled) state $|\psi\rangle \in H_{2^n}$.

Measurement. After application of a quantum circuit, the algorithm returns to classical mode (or another quantum circuit) by a measurement step in the computational basis of one or more qubits.

4 Simulating Quantum Algorithms by ASMs

An obvious, but minor, obstacle for the simulation of quantum algorithms by ASMs is that quantum algorithms are genuinely *probabilistic* algorithms, genuine in the sense that the probabilism is not simulated by a deterministic generation of pseudo-random numbers, but by the inherent probabilism in quantum mechanical processes. We thus have to assume that the simulating ASMs have access to true random numbers. Therefore, our quantum ASM will make use of a dynamic function **random** providing at each state access to a random real number in the interval $[0, 1]$.

The power of quantum computing comes from the possibility to process in one step a large number of entangled classical states. To put it differently, quantum algorithms go beyond classical concepts of sequential algorithm.

Proposition 1. *Quantum algorithms are not sequential algorithms in the sense of postulates* **(S1) - (S3)** *and can thus not be simulated by sequential ASM.*

Proof. Both unitary evolution steps and measurement steps violate the Uniformly Bounded Exploration Postulate. □

On the other side, it is not difficult to simulate quantum algorithms by parallel ASMs in the sense of [1] (we will make this precise below). But this is not the point. The point of quantum computing is to use the inherent parallelism in quantum effects for constructing more efficient algorithms than classical sequential algorithms. Indeed quantum algorithm can perform certain tasks (like searching databases or factoring integers) more efficiently than classical sequential algorithms can do. But we do not want to add anything else than "being quantum" to the paradigm of sequential algorithms.

A quantum state is a superposition of classical states. A computation step of a quantum algorithm changes a quantum state by modifying in parallel the underlying classical states (the base vectors of the Hilbert space). Hence it is clear that everything a quantum algorithm can do can also be done by a classical parallel algorithm. But quantum computers (if they can be built) implement exponential parallelism without using exponential ressources!

From the quantum point of view, quantum algorithms are sequential, the only parallelism being inside the basic operations of unitary transformation and measurement. An adequate simulation of quantum algorithms by ASMs should take this into account. Rather than going up to arbitrary parallel ASMs, we therefore define quantum ASMs as sequential ASMs, enriched by two kinds of rules that take care of the quantum operations.

States. Quantum ASMs must have states that are sufficiently rich to describe the mathematics of quantum gates and quantum measurement. For circuits over a fixed basis Ω of quantum gates, the background of the states should contain, along with the requirements given in [1], the basic arithmetic operations on complex numbers, natural numbers, binary strings, appropriate descriptions for the quantum gates in Ω (appropriately parametrised if Ω is infinite), and the like. In particular we assume the presence of basic operations on multisets, and a summation operator mapping any finite multiset M of complex numbers to the sum over its elements. (Formally, if $M \subseteq \mathbb{C}$ is given by $\mathrm{Mult}_M : \mathbb{C} \to \mathbb{N}$, then $\sum M = \sum_{z \in \mathbb{C}} \mathrm{Mult}_M(z) \cdot z$.)

Suppose the algorithm enters a quantum phase on n qubits, starting with a classically prepared vector $|x\rangle = |x_0 \dots x_{n-1}\rangle$ in the computational basis of H_{2^n}, and should perform a sequence of unitary operations of the type described in Definition 2. The corresponding state X of the algorithm must contain the information n and $|x\rangle$. The entangled states $|\psi\rangle$ that will be produced by the unitary evolution of the system can be desribed by a dynamic function $\Psi : \{0,1\}^* \to \mathbb{C}$ where

$$|\psi\rangle = \sum_{x \in \{0,1\}^n} \Psi(x)|x\rangle.$$

Given n and x the ASM initialises Ψ by $\Psi(x) = 1$ and $\Psi(y) = 0$ for $y \neq x$.

Unitary Transformations. To make precise how the operation

"apply U to the qubits $i_1, \dots, i_m$"

is described by an ASM rule observe that for every basis state $|x\rangle$ the relevant (i.e. non-zero) entries $U_{i_1 \dots i_m}(x, y)$ of the matrix $U_{i_1 \dots i_m}$ are those for which $x_k = y_k$ for all $k \notin \{i_1, \dots, i_m\}$. For $x \in \{0,1\}^n$, $i < n$ and $z \in \{0,1\}$, let $x[i/z]$ be the word obtained from x by replacing the i-th bit x_i by z. The application of $U_{i_1 \dots i_m}$ to the state defined by $\Psi : \{0,1\}^n \to \mathbb{C}$ is described by the rule

```
forall x ∈ {0,1}^n do
```
$$\Psi(x) := \sum \{\!\{ U(x_{i_1} \cdots x_{i_m}, z_1 \cdots z_m)\Psi(y) : z_1, \ldots, z_m \in \{0,1\}, y = x[i_1/z_1] \cdots [i_m/z_m] \}\!\}$$
```
enddo
```

Measurement. Given an entangled state $|\psi\rangle \in H_{2^n}$ and indices $i_1, \ldots, i_m$, measurement with respect to qubits $i_1, \ldots, i_m$ means computing the probabilities $p(u) = \langle \psi_u | \psi_u \rangle$ of the associated decomposition $|\psi\rangle = \sum_{u \in \{0,1\}^m} |\psi_u\rangle$, selecting one u according to this distribution, and projecting $|\psi\rangle$ to $\beta|\psi_u\rangle$.

We can implement this by selecting a random number $r \in [0,1]$, decomposing the unit interval into segments $S(u)$ (where $u \in \{0,1\}^m$) of length $p(u)$ and picking the u for which r is contained in $S(u)$. Again, this can be described in ASM-notation by a straightforward application of a **forall-do** rule and summation over a multiset. We assume that `random` is a nullary function whose value in each state is a random real number between 0 and 1. (At each fixed state the value of `random` of course is the same when it is called several times in parallel.)

```
forall x ∈ {0,1}^n, u ∈ {0,1}^m do
```
$$\texttt{if } \varphi(x,u) \texttt{ then } \Psi(x) := 1/\sqrt{p(u)}\Psi(x) \texttt{ else } \Psi(x) := 0$$
```
enddo
```

where

$$\varphi(x,u) := (\ell(u) \leq \texttt{random} < \ell(u) + p(u)) \wedge x_{i_1} = u_1 \wedge \cdots \wedge x_{i_m} = u_m$$
$$\ell(u) := \sum \{\!\{ p(v) : v \in \{0,1\}^m : v <_{\text{lex}} u \}\!\}$$
$$p(u) := \sum \{\!\{ \|\Psi(y)\|^2 : y \in \{0,1\}^n : y_{i_1} = u_1 \wedge \cdots \wedge y_{i_m} = u_m \}\!\}$$

5 Postulates for Quantum Algorithms

Having given a description of one (albeit rather general) model of quantum algorithms by ASMs the question arises if also in the quantum case, the ASM thesis can be derived from some basic postulates that all models for quantum computation satisfy.

We describe such postulates here. As in the case of sequential and parallel algorithms, we need the Sequential Time Postulate (S1) and the Abstract State Postulate (S2). The Uniform Bounded Exploration Postulate (S3) will have to be modified and we will need postulates decribing unitary transformation steps and

measurement steps. Concerning the probabilism inherent in quantum algorithms we follow the convention that probabilistic algorithms have a function `random` in their vocabulary that takes as values real numbers in the interval $[0, 1]$. Update sets $\Delta(A, X)$ that do not depend on the value of `random` are called deterministic, otherwise they are probabilistic. The value of `random` is supposed to be changed uniformly, independently and externally (i.e. not by the algorithm) at any step. Another possibility would be to modify the sequential time postulate so that the next step transformation is of the form $\tau_A : \mathcal{S}(A) \times [0, 1] \to \mathcal{S}(A)$.

We also need a variant of the Background Postulate, tailored for quantum algorithms, which gives some minimal requirements on what the states should contain, and which also imposes a closure condition.

Recall that a state of a computation should contain everything that is necessary to determine the future progress of that computation. Hence a state of a quantum algorithm does not merely consist of an element of an appropriate Hilbert space, but also of a classical state containing information on the unitary operations used, and of course also on the classical parts of the computation. Hence we essentially adapt the notion of state from [1, 3] by augmenting it with an element $|\psi\rangle$ of a Hilbert space. Sticking to the format of first-order structures, such an element is formally described by a function $\Psi : B \to \mathbb{C}$ from the basis B of the Hilbert space to the complex numbers. We use the notation $X = (X_c, |\psi\rangle)$ for the states of a quantum algorithm where X_c is the *classical part* of the state X (everything except Ψ).

Q1: Quantum State Postulate. Besides the conventions from [3], all states contain the three constants `classical`, `quantum`, and `measure` (which are interpreted by distinct elements) and a nullary function `mode` whose value is one of these three elements. Further all states contain a nullary function symbol `basis` whose value is a finite or countable set B, and a function symbol Ψ whose value is a function from B to the complex numbers $\mathbb{C}$, describing a unit length element $|\psi\rangle$ of the Hilbert space with orthonormal basis B. Further the state space $\mathcal{S}(A)$ of any quantum algorithm is closed under variations of $|\psi\rangle$, i.e., if $(X_c, |\psi\rangle) \in \mathcal{S}(A)$ then also $(X_c, |\varphi\rangle) \in \mathcal{S}(A)$ for every unit length $|\varphi\rangle$ of the same Hilbert space.

The function `mode` divides the state space into three classes, the classical states, the quantum states, and the measure states, each satisfying different postulates on the next step transformation.

The remaining postulates should ensure the following properties: For quantum states $(X_c, |\psi\rangle)$, the update of $|\psi\rangle$ is defined by a unitary transformation U (that depends only on X_c and A but not on $|\psi\rangle$). For measurement states $(X_c, |\psi\rangle)$ the update of $|\psi\rangle$ is defined by a (probabilistic) projection. Finally, the classical part X_c of any state $(X_c, |\psi\rangle)$ is updated by a sequential algorithm. The only unbounded parallelism of a quantum algorithm is the simultaneous update of all amplitudes in unitary transformation and measurement steps.

Remark. One might criticise this last requirement as too restricted: why not consider "quantum parallel algorithms"? The answer is that the very point of

quantum algorithms is to use quantum effects in order to implement with polynomial ressources what classically requires exponential parallelism. Of course one could define computation models that combine quantum computing with unbounded classical parallelism, but this seems not interesting since we lose the advantage of quantum algorithms without going beyond the power of classical parallel algorithms.

It is relatively straightforward to formulate postulates for the first two properties.

Q2: Unitary Transformation Postulate. If $X = (X_c, |\psi\rangle)$ is a quantum state then $\tau_A(X)$ is either a quantum or a measure state, with the same value of `basis`. The change of $|\psi\rangle$ from X to $\tau_A(X)$ is described by a unitary operator U_X (which only depends on X_c).

Q3: Measurement Postulate: If $X = (X_c, |\psi\rangle)$ is a measure state then the change of $|\psi\rangle$ from X to $\tau_A(X)$ is decribed by a collection $\{M_{X,j} : j \in J\}$ of linear operators (which only depends on X_c) such that $\sum_{j\in J} M^*_{X,j} M_{X,j} = I$ and the index set J is linearly ordered.

The value of $|\psi\rangle$ after the measurement step is described by

$$\frac{M_{X,j}|\psi\rangle}{\sqrt{\langle\psi|M^*_{X,j}M_{X,j}|\psi\rangle}}$$

for one j, the probability for each j being $p(j) = \langle\psi|M^*_{X,j}M_{X,j}|\psi\rangle$.

Remark. When we say that a linear operator W defining the update of $|\psi\rangle$ in a state $X = (X_c, |\psi\rangle)$ depends only on X_c, we mean that the same operator W applies to every $X' = (X_c, |\psi'\rangle)$ with the same classical part X_c.

It is a little more delicate to formulate in a precise, yet natural way that the only parallelism is inside unitary transformation and measurement steps. In fact there are several possibilities.

We split the update sets of a quantum algorithm A on state $X = (X_c, |\psi\rangle)$ into the classical updates of the functions in X_c and the updates of $|\psi\rangle$, i.e., $\Delta(A, X) = \Delta(A, X_c) \cup \Delta(A, |\psi\rangle)$. A high-level formulation of the sequentiality postulate might be the following.

Q4: Sequentiality Postulate. For every quantum algorithm A, the classical update sets and all entries of the unitary transformation matrices and the projection matrices at quantum states and measure states are computed by sequential algorithms.

Here is a more explicit version of postulate Q4.

Q4': Sequentiality Postulate (Detailed Version). For every quantum algorithm A, there exist sequential algorithms A_1, A_2, A_3 with the following properties:

(1) A_1 has the same states as A and at every state $X = (X_c, |\psi\rangle)$ of A, the classical update set $\Delta(A, X_c)$ equals $\Delta(A_1, X)$, and in the case that X is a classical state, $\Delta(A_1, X)$ is the entire update set $\Delta(A, X)$,
(2) A_2 has states (X_c, x, y) consisting of the classical part of a quantum state X of A, and base vectors x, y in `basis`, and the set of outputs $out_{A_1}((X_c, x, y))$ contains exactly the entry $U_X(x, y)$ of the unitary transformation U_X.
(3) A_3 has states (X_c, x, y, j) consisting of the classical part of a measure state X of A, base vectors x, y in `basis`, and $j \in J$, and the set of outputs $out_{A_1}((X_c, x, y, j))$ contains exactly the entry $M_{X,j}(x, y)$ of the projection matrix $M_{X,j}$ at state X.

The same postulate can also be formulated in a different way, similar to the Uniformly Bounded Exploration Postulate for sequential algorithms.

Q4": Limited Exploration Postulate. Let $\Upsilon = \Upsilon_c \cup \{\Psi\}$ be the vocabulary of the quantum algorithm A. There exist finite sets T_1, T_2, T_3 of terms with the following properties.
(1) T_1 is a set of ground terms over vocabulary Υ and whenever two states X and X' coincide on T_1, then $\Delta(A, X_c) = \Delta(A, X'_c)$, and if X and X' happen to be classical states, then even $\Delta(A, X) = \Delta(A, X')$.
(2) T_2 has vocabulary Υ_c and for all $t \in T_2$, free$(t) \subseteq \{x, y\}$. Whenever X and X' are two quantum states of A, a, b are elements of `basis` in X, and a', b' are elements of `basis` in X', such that (X_c, a, b) and (X'_c, a', b') coincide on T_2, then the entries $U_X(a, b)$ and $U_{X'}(a', b')$ of the appropriate unitary transformations are equal.
(3) T_3 has vocabulary Υ_c and for all $t \in T_2$, free$(t) \subseteq \{x, y, j\}$. Whenever X and X' are two measure states of A, a, b are elements of `basis` in X, and a', b' are elements of `basis` in X', such that (X_c, a, b, k) and (X'_c, a', b', k') coincide on T_3, then the entries $M_{X,k}(a, b)$ and $M_{X',k'}(a', b')$ of the appropriate projections are equal.

Example: Quantum Turing Machines. One of the popular models in the theory of quantum algorithms are quantum Turing machines (QTMs). For details, see e.g. [2, 5]. We verify that QTMs satisfy our postulates. A *quantum Turing machine (QTM)* is a tuple $M = (Q, \Sigma, q_0, q_a, q_r, \delta)$ with alphabet Σ, finite set of control states Q, initial state q_0, accepting state q_a, rejecting state q_r, and a transition function $\delta : Q \times \Sigma \times \Sigma \times Q \times \{-1, 0, 1\} \to \mathbb{C}$, where $\delta(q, \sigma, , q', \sigma', m)$ defines the amplitude that whenever the machine is in state q and reads symbol σ, it replaces σ with σ', enters state q' and moves the head in direction m. The transition function must satisfy the condition that the induced transformation on the configuration space is unitary.

As in the case of classical Turing machines, a configuration of a QTM is determined by the content of the tape, the current state, and the position of the head. Let C_M denote the set of all configurations of M and let H_M be

the Hilbert space with orthonormal basis $\{|c\rangle : c \in C_M\}$. The transition function δ induces a mapping $a : C_M \times C_M \to \mathbb{C}$ giving for every pair (c, c') of configurations the amplitude of the transition of M from the basis state $|c'\rangle$ to $|c\rangle$. The *quantum evolution* defined by M on H_M maps $|\psi\rangle = \sum_{c \in C_M} \Psi(c)|c\rangle$ to $U_M|\psi\rangle = \sum_{c \in C_M} \sum_{c' \in C_M} a(c, c')\Psi(c')|c\rangle$. The standard measurement is the measurement with respect to the basis $\{|c\rangle : c \in C_M\}$ (at the end of the computation).

Recall that the states of an algorithm are supposed to contain everything that is necessary to determine the future progress of a computation. For a QTM M this means that the states contain the static function `basis` whose value is the set of configurations C_M. Further the states contain functions `state`: $C_M \to Q$, `content`: $C_M \times \mathbb{Z} \to \Sigma$, `head`: $C_M \to \mathbb{Z}$ and a relation `replace`$\subseteq C_M \times \Sigma \times Q \times \{-1, 0, 1\} \times C_M$ describing the configurations and the cross-linking between the configurations (namely, `replace`(c, σ, q, z, c') holds iff c' results from c by replacing the contents of the `head`(c)-th cell of c with σ, the state of c with q and moving the head according to z).

The sequential algorithm computing for every state X the entries $U_X(c, c')$ of the associated unitary transformation returns $\delta(q_1, \sigma_1, \sigma_2, q_2, d)$ in the case that `replace`$(c', \sigma_2, q_2, d, c)$ holds where $\sigma_1 =$ `content`$(c', $`head`$(c'))$ and $q_1 =$ `state`(c'). Otherwise, it returns 0. Note that for each c there are only finitely many non-zero values $U(c, c')$. Concerning measurement steps in a quantum Turing machine, the set J in the postulate Q3 is C_M. The sequential algorithm demanded in postulate Q4', returns for a basis vector b the value 1 if it is in the subspace spanned by the basis vector v and 0 otherwise.

6 The Equivalence Theorem

The following lemma follows easily from the definition of sequential ASMs.

Lemma 1. *For every sequential ASM rule there exists an equivalent ASM rule of the form*

```
do-in-parallel
   if φ1 then R1
   if φ2 then R2
       ⋮
   if φn then Rn
enddo
```

where R_i is an update rule of the form $s_i := t_i$ or an output-rule `Output`(t_i).

Theorem 1. *Every quantum algorithm is equivalent to an ASM working with the same background.*

Proof. As already mentioned the states of a quantum algorithm can be partitioned into three classes: classical states, quantum states and measurement states.

Classical states. Postulate Q4' and the sequential ASM thesis imply that there exists an ASM A_c whose update set equals the classical update set of the quantum algorithm on any of its states. In the case of a classical state, the update set of A_c is the entire update set.

Quantum states. Postulate Q4' and the sequential ASM thesis for ASMs with output imply that there exists an ASM whose output for every quantum state X and every pair of base vectors $b, c \in \texttt{basis}$ contains exactly the entry $U_X(b, c)$. Further, there is an equivalent ASM with a program of the form given in Lemma 1 and with $\text{free}(\varphi_i), \text{free}(t_i) \subseteq \{b, c\}$. Since we are only interested in the output we can assume that all rules R_i, $i \in \{1, ..., n\}$ are output-rules of the form $\texttt{Output}(t_i)$. The following ASM A_q simulates the execution of the unitary transformation induced by the quantum algorithm in the case of a quantum state (w.l.o.g., we assume that $\bigvee_{i=1}^{n} \varphi_i$ is valid and for all $k, l \in \{1, ..., n\}, k \neq l$ the formula $\varphi_k \wedge \varphi_l$ is not satisfiable):

$$\begin{array}{l}\texttt{forall } b \texttt{ : } b \in \texttt{basis do}\\ \quad \Psi(b) := \sum_{i=1}^{n} \sum \{\!\{ t_i(b, c) \cdot \Psi(c) : c \in \texttt{basis} : \varphi_i(b, c) \}\!\}\\ \texttt{enddo}\end{array}$$

Measure States. With analogous arguments, we infer that there exists an ASM with a program of the form given in Lemma 1 whose output for every measure state X, every pair of base vectors $b, c \in B$ and every $j \in J$ contains exactly the entry $M_{X,j}(b, c)$. The rules R_i, $i \in \{1, ..., n\}$, are output-rules of the form $\texttt{Output}(t_i)$ and $\text{free}(\varphi_i), \text{free}(t_i) \subseteq \{b, c, j\}$. The following ASM A_m simulates the measurement induced by the quantum algorithm in the case of a measure state and an already chosen value for $\texttt{random}$.

$$\begin{array}{l}\texttt{forall } b, i \texttt{ : } b \in \texttt{basis} \wedge i \in J \texttt{ do}\\ \quad \texttt{if } \ell(i) \leq \texttt{random} < \ell(i) + p(i) \texttt{ then}\\ \quad\quad \Psi(b) := \frac{1}{\sqrt{n(i)}} \sum_{j=1}^{n} \sum \{\!\{ t_j(b, c) \cdot \Psi(c) : c \in \texttt{basis} : \varphi_j(b, c, i) \}\!\}\\ \texttt{enddo}\end{array}$$

where

$\ell(j)$ is the probability that the subspace belonging to a k with $k < j$ is measured (the set J is linearly ordered by postulate Q3).

$p(j)$ is the probability that the subspace belonging to j is measured.

$n(j)$ is the inverse of the square of the renormalization factor. It is obtained by summing the $\Psi_{\text{new}}(b)^* \cdot \Psi_{\text{new}}(b)$ for all base-vectors b where Ψ_{new} is the amplitude function after executing the transformation M_j and before carrying out normalization.

Hence $\ell(j), p(j), n(j)$ can be defined as follows:

$$\begin{aligned}\ell(j) &= \sum \{\!\{ p(i) :: i \in J \wedge v < j \}\!\}\\ p(j) &= \sum \{\!\{ \Psi(b)^* \cdot \Psi_{\text{new}}(b) : b \in \texttt{basis} : \mathit{true} \}\!\}\\ n(j) &= \sum \{\!\{ \Psi_{\text{new}}(b)^* \cdot \Psi_{\text{new}}(b) : b \in \texttt{basis} : \mathit{true} \}\!\}\end{aligned}$$

with $\Psi_{\text{new}}(b) = \sum_{j=1}^{n} \sum \{\!\{ t_j(b,c) \cdot \Psi(c) : b \in \texttt{basis} : \varphi_j(b,c,j) \}\!\}$. The desired ASM combines A_c, A_q, and A_m as follows:

```
do-in-parallel
   A_c
   if mode = quantum then A_q
   if mode = measure then A_m
enddo
```

This completes the proof of the theorem. □

Remark. From the proof of Theorem 1, we see that it is not necessary to allow arbitrary parallelism in the ASMs that simulate quantum algorithms. The classical update sets are generated by a sequential ASM. Unbounded parallelism does appear in the simulation of the quantum steps and the measure steps, but in a very restricted way. We have only one forall-instruction (without nestings) and an update rule envolving a uniformly bounded number of summations over a multiset. Comprehension terms are not nested (the ASM given in the proof uses nested comprehension terms but it can be rewritten without nestings). Hence, the equivalence theorem could be formulated in a stronger way by allowing only ASMs satisfying the above constraints.

There are also many cases where multisets are not needed. One example are quantum Turing machines. In this case, every base vector (i.e. every configuration) is connected via a non-zero entry of the transoformation matrix to a uniformly bounded number of other base vectors (depending only on the algorithm, not on the input). Summations over the multisets for the quantum states can thus be replaced by sums over a uniformly bounded number of elements.

References

[1] A. Blass and Y. Gurevich, *Abstract state machines capture parallel algorithms*, Tech. Rep. MSR-TR-2001-117, Microsoft Research, 2001. 310, 312, 315, 316, 318

[2] J. Gruska, *Quantum Computing*, McGraw-Hill, 1999. 312, 320

[3] Y. Gurevich, *Sequential Abstract State Machines capture sequential algorithms*, ACM Transactions on Computational Logic, 1 (2000), pp. 77–111. 309, 310, 311, 318

[4] Y. Gurevich, *Logician in the land of OS: Abstract State Machines in Microsoft*, in Proc. 16th IEEE Symp. on Logic in Computer Science, 2001, pp. 129–136. 309

[5] M. Hirvensalo, *Quantum Computing*, Springer, 2001. 312, 320

[6] M. Nielsen and I. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, 2000. 312

[7] P. Shor, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, SIAM Journal on Computing, 26 (1997), pp. 1484–1509. 310

[8] L. Vandersypen, M. Steffen, G. Breyta, C. Yannoni, M. Sherwood, and I. Chuang, *Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance*, Nature, 414 (2001), pp. 883–887. 310

Consistent Integration for Sequential Abstract State Machines

Marianna Nicolosi Asmundo and Elvinia Riccobene

Dipartimento di Matematica e Informatica, Università di Catania, Italy
{nicolosi,riccobene}@dmi.unict.it

Abstract. In this paper we deal with the problem of integrating components described as Sequential Abstract State Machines. Two operations are defined to compose sequential ASMs and returning a machine consistent with respect to updates. Due to their nature, such operations can be used to analyze and handle "behavioral inconsistencies" and to prove component and system properties. The applicative aspects of the theoretical issues introduced in the paper are shown by means of three examples: the behavioral description of a telephone system presented in [7], the ASM specification of the Production Cell case study developed in [3], and the ASM solution of the railroad crossing problem given in [12].

1 Introduction

Modelling complex systems in a coherent and rigorous way is a basic goal in system design. In particular, the integration of specifications could be a problematic operation having potentially severe side effects: when components of a system separately developed are combined together, they may intersect and overlap giving rise to inconsistencies and conflicts which have to be resolved in order to obtain a consistent system. Moreover, the substitution of old components with new ones or the introduction of new features in the system may affect the overall system behavior in an unpredictable way. For the reasons above, the introduction of composition operations producing consistent systems is required.

Here, we address the problem for Abstract State Machines focusing on the composition of sequential ASMs. The choice of this formal method as a platform to construct rigorous integration operations for partial specifications is intentional and it is due to the fact that, besides having been successfully used in practice for design and analysis of complex hardware/software systems [5], ASMs have evident theoretical foundations, clear and precise semantics [10, 11, 2, 15].

An abstract state machine may be employed to describe the behavior of a system component, of a process to be integrated with other processes, etc. Having a rigorous operation for the integration of ASMs guarantees that we are able to construct the behavioral specification of a device from its features or to build a system from its components without flatten them and preserving the interconnections between them.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 324–340, 2003.
© Springer-Verlag Berlin Heidelberg 2003

When using components to describe a system behavior, components may represent different features of the same device, or different subsystems of a compound system. Therefore, in this paper we introduce two kinds of composition operations over sequential ASMs. The first one, called *feature composition* and individuated by the symbol $\oplus$[1], takes as input sequential ASMs representing features of a device and returns a sequential ASM being the required device. The second one, called *component composition* and individuated by the symbol $\otimes$, takes as input sequential ASMs representing components of a system. In case the components are synchronous, i.e. they have the same clock, the result of the operation is a multi-agent ASM with synchronous agents, otherwise, the result is a multi-agent ASM with asynchronous agents.

Operations $\oplus$ and $\otimes$ are both equipped with conditions of compatibility for the components to be integrated, and of tests to check the consistency of the compound system with respect to updates under the assumption that the components are already consistent with respect to updates. Such tests allow us to find and recognize possible update inconsistencies coming out from the integration process.

In section 2 we introduce some preliminary concepts on ASMs. In section 3 we describe three applications, which we later use to show some applicative results of the theoretical issues introduced in this paper. In sections 4, 5 and 6, the composition operations $\oplus$, $\otimes$ for synchronous components and $\otimes$ for asynchronous components are respectively defined; their application to our case studies is also presented. In section 7, it is shown how the composition operations can be applied to analyze and handle behavioral inconsistencies and to prove system properties. Finally, section 8 concludes giving some hints of future work.

2 Preliminaries

In this section, we introduce some notions which will be used throughout the paper in order to illustrate the composition operations $\oplus$ and $\otimes$ over sequential ASMs resulting, respectively, in sequential and multi-agent ASMs.

2.1 Sequential Abstract State Machines

The formal definition of Sequential Abstract State Machines was given by Gurevich in [11] where the thesis claiming that every sequential algorithm can be simulated by a sequential ASM is thoroughly treated. We refer the reader to [11] for the basic definitions and concepts concerning Sequential Abstract State Machines.

2.2 Multi-agent Abstract State Machines

The integration of system components may return multi-agent ASMs (see definition in [10]) with synchronous agents, i.e. agents running in parallel and having

[1] The same symbol is used in [6] to define the operator *seq*, but the two operations have different semantics.

the same clock, or with asynchronous agents, i.e. agents proceeding in parallel at their own speed and whose communications may provide the only logical ordering between their actions. A multi-agent ASM with synchronous agents is a multi-agent ASM with quasi-sequential runs, while a multi-agent ASM with asynchronous agents corresponds to a multi-agent ASM with partially ordered runs (definitions in [10]). We slightly modify here the definitions of quasi-sequential runs and partially ordered runs, given in [10], to formalize agents interacting with the environment. We treat the distinct cases under the assumption that every agent moves as soon as it is enabled.

Run of a Multi-agent ASM with Synchronous Agents. Similarly to the definition of run for sequential ASMs interacting with the environment [11], moves of the synchronous agents of a multi-agent ASM may be intermixed with moves of the environment, so there may be sequences of moves of kind $\mu_1, \nu_1, \mu_2, \nu_2, \ldots$ where each μ_i is constituted by the moves of all the agents of the system which are enabled in the current state and ν_i are executed by the environment.

Definition 1. *A run of a multi-agent Abstract State Machine M with synchronous agents, is a quasi-sequential run $X_0, X_1, X_1', X_2, X_2', \ldots$ where X_0 is an initial state, every X_i' is obtained from X_i by an action of the environment and every X_j is obtained from the previous state by firing in parallel the programs of the agents of M.*

Run of a Multi-agent ASM with Asynchronous Agents. When synchronism of the sequential ASM components is not required, quasi-sequential runs cannot describe the behavior of the composed system, we consider the more generic runs described by partially ordered sets.

In this paper we restrict our attention to the case where moves are atomic, states are global, agents are deterministic and we allow the presence of moves performed by the environment. Our definition is obtained by slightly modifying the definition of partially ordered run in [10]:

Definition 2. *A* run *ρ of a multi-agent ASM $\mathcal{M}$ with asynchronous agents can be defined as a triple (M, A, σ) satisfying the following conditions:*

1. *M is a partially ordered set of moves, where all sets $\{y : y \leq x\}$ are finite. Elements of M represent moves made by various agents and by the environment during the run. If $y < x$ then x starts when y is already finished; that explains why the set $\{y : y \leq x\}$ is finite.*
2. *A is a function on M such that every non empty set $\{x : A(x) = a\}$ and $\{x : A(x) = e\}$, where a is an agent of $\mathcal{M}$ and e is the environment, is linearly ordered.*
3. *σ assigns a state of $\mathcal{M}$ to the empty set and to each initial segment of M; $\sigma(\emptyset)$ is an initial state; $\sigma(X)$ is the result of performing all moves in X.*
4. Coherence condition*: If x is a maximal element in a finite initial segment X of M and $Y = X - \{x\}$ then either:*

- $A(x) = a$, *where* a *is an agent in* $\sigma(Y)$ *and* $\sigma(X)$ *is obtained from* $\sigma(Y)$ *by firing* $A(x)$ *at* $\sigma(Y)$, *or*
- $A(x) = e$ *and* $\sigma(X)$ *is obtained from* $\sigma(Y)$ *by the execution of the move* x *over* $\sigma(Y)$.

Intuitively, a run can be seen as the common part of histories of the same computation recorded by various observers.

3 Case Studies

We introduce three examples, the Telephone System, the Production Cell and the Railroad Crossing, used later as application cases of the composition operations $\oplus$ and $\otimes$.

3.1 Telephone System

This example is drawn from the behavioral specification of a telephone. It was introduced by Easterbrook and Nuseibeh in [7] in order to illustrate a methodology to handle specification inconsistencies using the notion of ViewPoint as defined in [9], and the original formalization was in Harel's statecharts.

The system can be modeled by means of two ASMs, one describing the behavior of a calling telephone and the other describing the behavior of a called telephone. The calling telephone can be in *mode* Idle (the stand-by mode), Dialtone, Ringtone, Connected, Busytone, and the telephone state can change under the events: *Lift, Replace, Calleelift, Calleereplace, Dialfree, Dialbusy.* The receiving telephone has *mode* Idle, Dialtone, Ringing, Connected, and its state may change under the events *Callerdial, Callereplace, Lift, Replace.*

Caller program

R1 = **if** *mode* = Idle **and** *Lift* **then** *mode*:= Dialtone

R2 = **if** *mode* ≠ Idle **and** *Replace* **then** *mode*:= Idle

R3 = **if** *mode* = Dialtone **then**
if *Dialfree* **then** *mode*:= Ringtone **else if** *Dialbusy* **then** *mode*:= Busytone

R4 = **if** *mode* = Ringtone **and** *Calleelift* **then** *mode*:= Connected

R5 = **if** *mode* = Connected **and** *Calleereplace* **then** *mode*:= Dialtone

Callee program

R1′ = **if** *mode* = Idle **and** *Callerdial* **then** *mode*:= Ringing

R2′ = **if** *mode* = Ringing **and** *Callereplace* **then** *mode*:= Idle

R3′ = **if** *mode* = Dialtone **and** *Replace* **then** *mode*:= Idle

R4′ = **if** *mode* = Ringing **and** *Lift* **then** *mode*:= Connected

R5′ = **if** *mode* = Connected **and** *Callereplace* **then** *mode*:= Dialtone

Despites the extreme simplicity of the example, we find this specification significant since, being symmetric – Caller and Callee specifications can be seen as two features of the same device or as two interacting devices –, it lends itself to be an application of both compositions $\oplus$ and $\otimes$.

3.2 Railroad Crossing

The railroad crossing problem is a benchmark case study used to compare various formal methods and to better understand their application in developing practical real-time systems. Here we consider the ASM solution of such problem given in [12]. The system is specified as a multi-agent ASM with two synchronous modules: the `gate` and the `controller`. The `gate`'s job is to open and close the gate of the crossing while the `controller`, after checking whether there is a train coming, or leaving the crossing, determines the `gate` component to close or open the gate. The ASM specifying the railroad crossing is highly parallel in the sense of [2], in fact the number of tracks of the crossing depends on the initial state. Since in this paper we are interested in sequential machines, we restrict our attention to the case of a railroad crossing with a fixed number of tracks.

3.3 Production Cell

The production cell control problem was posed in [13] as a case study derived from an "actual industrial installation in a metal-processing plant in Karlsruhe" to obtain a "realistic, comparative survey" for testing "the usefulness of formal methods for critical software systems and to prove their applicability to real world examples"[14]. Börger and Mearelli propose a solution to the production cell control problem in [3], show how to integrate the use of ASMs into a complete software development life cycle and, of particular interest to our work, proved safety and liveness properties of the system. *... the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a travelling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane*[14]. The system is specified "as a multi-agent ASM with six modules, one for each *agents*" – the `Feed Belt`, the `Robot`, the `Press`, the `Deposit Belt`, the `Travelling Crane`, the `Elevating Rotary Table` – "composing the production cell, and working together concurrently where each of the component ASMs follows its own clock. Each of the agents represents a sequential process which can execute its rule as soon as it becomes enabled".

`Robot`'s Tasks. Among production cell components, the `Robot` is the only one composed by four distinct machines describing its tasks, namely: (1) to pick up a metal blank from the elevating rotary table; (2) to pick up a forged piece from the press; (3) to put a piece on the deposit belt; (4) to put a metal blank in the press. To prove properties of the system and to study the interconnection between the `Robot` and other components, sometimes it is more efficient to analyze the subcomponents of the `Robot` separately. Their $\oplus$-composition gives the entire specification of the `Robot`.

4 Composing Distinct Behaviors of a Same Device

Our first case takes as input n sequential ASMs $M_1, \ldots, M_n$ which describe the behavior of different features of a same device. According to the formal definition in [11], every M_i is provided with: a vocabulary Υ_i, a program Π_i, a set $\mathcal{S}(M_i)$ of Υ_i-structures closed under isomorphism and under the map τ_{Π_i}, a set $\mathcal{I}(M_i) \subseteq \mathcal{S}(M_i)$ that is closed under isomorphism, the map τ_{M_i} which is the restriction of τ_{Π_i} to $\mathcal{S}(M_i)$. We classify dynamic function symbols in Υ_i according to [4] as follows:

- $F_{Mon}(M_i)$ is the class of functions whose value can only be read by M_i and which describe events external to M_i and, in general, controlled by the environment;
- $F_{Ctrl}(M_i)$ is the class of functions that can be read and written by M_i only;
- $F_{Out}(M_i)$ is the class of functions that can be written by M_i and which describe an output from M_i to the environment;
- $F_{Sh}(M_i)$ is the class of functions that describe communication between different agents, they can be read and written by M_i.

While an interactive ASM may have function symbols from any of these classes, a non interactive ASM has just controlled functions. We define the composition operation for interactive ASMs, being non interactive ASMs a particular case where $F_{Mon}(M_i) = F_{Out}(M_i) = F_{Sh}(M_i) = \emptyset$.

Compatibility and Inconsistencies. Before introducing the composition operation, we give a condition necessary to apply the operation to a collection of sequential ASMs, together with some *updates consistency tests*.

We admit the application of our composition operation over a collection of sequential ASMs only if they are *compatible* in the sense of the definition below:

Definition 3. *Let $M_1, \ldots, M_n$ be sequential ASMs. They are* compatible *if for every $f \in \Upsilon_i \bigcap \Upsilon_j$, $i, j \in \{1, \ldots, n\}$, $i \neq j$, if dynamic, f belongs to the same function class in M_i and in M_j.*

Note that a function symbol f not satisfying the conditions of definition 3 cannot have a clear semantics in the composed machine. However, by suitably renaming functions, in some cases it is possible to make components compatible.

Even if the components are compatible, the composition may be not consistent with respect to the updates. In fact, components interaction represented by dynamic functions that components share, may cause inconsistencies leading to unpredictable results.

Definition 4. *A sequential ASM M is* consistent with respect to updates *if for each state $X \in \mathcal{S}(M)$, if $(f, \bar{a}, b_1)$, $(f, \bar{a}, b_2) \in \Delta(M, X)$ then $b_1 = b_2$, being $f \in \Upsilon$, $\bar{a}, b_1, b_2$ elements of X, $\Delta(M, X)$ the set of function updates that X originates over M.*

Suppose that $M_1, \ldots, M_n$ are consistent with respect to updates according to the definition above. We need to check if their composition – which will result in a sequential ASM (see below the definition of $\oplus$-composition) – is consistent with respect to updates. We introduce some consistency tests that, under certain conditions guarantee that the composition is consistent. If the first test, called *safe test*, succeeds, we can construct a compound ASM which is *consistent with respect to updates*. Otherwise, we cannot answer about the consistency of the compound ASM just by inspection of the components, and we try another test, called *unsafe test*, applicable to the compound sequential ASM.

Safe Test. For every couple of indexes $i, j \in \{1, \ldots, n\}$, $i \neq j$, and for every $f \in (F_{Out}(M_i) \bigcup F_{Ctrl}(M_i) \bigcup F_{Sh}(M_i)) \bigcap (F_{Out}(M_j) \bigcup F_{Ctrl}(M_j) \bigcup F_{Sh}(M_j))$ do the following:

- let $R_i^1, \ldots, R_i^l$ be all the rules in M_i updating f,
- let $R_j^1, \ldots, R_j^m$ be all the rules in M_j updating f,
- let g_i^h, $h \in \{1, \ldots, l\}$, and g_j^k, $k \in \{1, \ldots, m\}$, be the guards of the rules above.

If $\varphi = \bigvee_{h,k}(g_i^h \wedge g_j^k)$ is a ground contradictory formula, then there are no simultaneous updates for f.

If this test fails, it means that the formula φ is satisfiable. The next step is to check whether the compound system $M = \oplus_{i=1}^n M_i$ (see definition 5) is not a model for φ. This will be achieved by means of the following unsafe test (to be performed by hand or by automated theorem provers):

Unsafe Test. Check whether the composition M is a model for φ or not.

If $M \not\models \varphi$ then there are not simultaneous updates of the same function (composition consistent with respect to updates); if $M \models \varphi$ then there may be updates inconsistencies coming out during the runs of M. In the latter case, it is necessary to analyze those parts of the specification which lead to failure of the test in order to check if the composition is still possible or not.

Definition 5 ($\oplus$-Composition for Features of a Same Device). Given n compatible sequential ASMs $M_1, \ldots, M_n$ with vocabularies $\Upsilon_1, \ldots, \Upsilon_n$, the $\oplus$-composition of $M_1, \ldots, M_n$ is the sequential ASM $M = \oplus_{i=1}^n M_i$ constructed as follows:

1. the vocabulary of M is $\Upsilon = \bigcup_{i=1}^n \Upsilon_i$ and the classes of dynamic functions of M are constructed by the function classes of the components in this way: $F_{Class}(M) = \bigcup_{i=1}^n F_{Class}(M_i)$, for $Class = Mon, Ctrl, Out, Sh$;
2. the program of M is Π: **do in parallel** $\Pi_1 \ \ldots \ \Pi_n$, where $\Pi_1, \ldots, \Pi_n$ are respectively the programs of $M_1 \ldots, M_n$;
3. the set of states, $\mathcal{S}(M)$, is a set of Υ-structures X constructed from a (not necessarily unique) n-uple $X_1, \ldots, X_n$ of states of $M_1, \ldots, M_n$ such that:
   - there exists an i, $i \in \{1, \ldots, n\}$, such that X_i is the reduct of X to Υ_i,
   - and for every $f \in \Upsilon \setminus \Upsilon_i$, f is interpreted in X as in X_j for some j such that $f \in \Upsilon_j$;

$\mathcal{S}(M)$ is closed under isomorphism;
4. the set of initial states $\mathcal{I}(M) \subseteq \mathcal{S}(M)$ is constructed from $\mathcal{I}(M_i)$ (as in step 3.); $\mathcal{I}(M)$ is closed under isomorphism;
5. the one-step transformation of M is the map $\tau_M : \mathcal{S}(M) \to \mathcal{S}(M)$ such that $\tau_M(X) = X + \Delta(M, X)$, where $\Delta(M, X) = \bigcup \Delta(M_i, X_i)$, X_i being the states constructing X; τ_M is a restriction of τ_Π to $\mathcal{S}(M)$.

For lack of space we skip here proofs and details of 3, 4, 5.

The $\oplus$-composed ASM M is a sequential algorithm, in fact it satisfies the Sequential Time, Abstract State and Bounded Exploration postulates (see [11]).

4.1 Case Study: Composing two Features of a Telephone

We examine here the composition of Caller and Callee as two features of the same device. Trivially they satisfy the compatibility condition. The application of the consistency tests gives the following results: (a) the safe test fails due to several not contradictory disjuncts in φ, for example ($mode \neq$ Idle **and** *Replace*) $\wedge$ ($mode =$ Ringing **and** *Lift*) which is the conjunction of the guard of rule R2 of the Caller and of the guard of rule R4′ of the Callee; (b) the unsafe test succeeds for all the disjuncts but one ($mode \neq$ Idle **And** *Replace*) $\wedge$ ($mode =$ Dialtone **And** *Replace*) which is satisfied by the compound system and corresponds to the conjunction of the guard of rule R2 of the Caller and of the guard of rule R3′ of the Callee. Looking directly at those rules, we see that firing them at the same time never leads to inconsistencies, so we can consistently compose the specifications obtaining a device holding the features of giving and receiving phone calls. The device can be enriched with other features in the same way.

4.2 Case Study: Integrating Robot's Components

Here we examine the $\oplus$-composition of the sequential ASMs describing distinct functionalities of the `Robot`. For space reasons we don't give their ASM code which can be found in [3]. It is easy to check that the four subcomponents of the `Robot` satisfy the compatibility condition. Regarding consistency with respect to updates, the only functions that, updated in two or more subcomponents, may be problematic are the controlled function *currPhase*, which is updated by all the subcomponents, and the output function *PressLoaded*, updated by two subcomponents. The application of the safe test to both functions is successful and so we can consistently compose all the subcomponents obtaining the specification of the `Robot`.

5 Composition of Distinct System Components Moving Synchronously

In our second case, we take as input n sequential ASMs $M_1, \ldots, M_n$ which describe the behavior of system components and all having the same clock.

Compatibility and Inconsistencies. Before introducing the composition operation for synchronous components, we give the compatibility condition to apply the operation, together with some updates consistency tests. Since we open the components to the outside world, we slightly modify them by parameterizing rules with *Self* and by refining the interaction functions if needed.

Let $M_1, \ldots, M_n$ be the components modified as above, $\Upsilon_1, \ldots, \Upsilon_n$ their vocabulary and $\Pi_1, \ldots, \Pi_n$ their programs. We apply the composition operation over the collection $M'_1, \ldots, M'_n$ where $M'_i = M_i[Self \leftarrow a_i]$ (substitute the function symbol *Self* with the name of the agent a_i), $\Upsilon'_i = \Upsilon_i \setminus \{Self\}$ and $\Pi'_i = \Pi_i[Self \leftarrow a_i]$ for $i = 1, \ldots, n$. At first, we check whether the components are compatible according to the definition below:

Definition 6. *$M'_1, \ldots, M'_n$ are* compatible *if for every $f \in \Upsilon'_i \bigcap \Upsilon'_j$, $i, j \in \{1, ..., n\}$, $i \neq j$, f has the same arity in Υ'_i and in Υ'_j.*

Supposing that the components are compatible and that each one is consistent with respect to updates, we have to make sure that their composition results in a multi-agent ASM which is consistent with respect to updates according to the following:

Definition 7. *A multi-agent ASM M with synchronous agents is* consistent *with respect to updates if no Υ-state X of M originates two updates $(f, \bar{a}, b_1)$ and $(f, \bar{a}, b_2)$ such that $b_1 \neq b_2$, being $f \in \Upsilon$ vocabulary of M, $\bar{a}, b_1, b_2$ elements of X.*

At first we apply the following preliminary test:

Function Test. If the following conditions are satisfied, then the composition of $M'_1, \ldots, M'_n$ is *consistent* with respect to updates:

- for every $f \in \bigcup_{i=1}^{n} \Upsilon'_i$, if $f \in F_{Ctrl}(M'_i)$, $i \in \{1, \ldots, n\}$, then $f \notin (F_{Ctrl}(M'_j) \bigcup F_{Sh}(M'_j) \bigcup F_{Mon}(M'_j) \bigcup F_{Out}(M'_j))$ for every $j \in \{1, \ldots, n\}$, $i \neq j$;
- $(F_{Out}(M'_i) \bigcup F_{Sh}(M'_i)) \bigcap (F_{out}(M'_j) \bigcup F_{sh}(M'_j)) = \emptyset$ for every $i, j \in \{1, ..., n\}$, $i \neq j$.

If this test fails we can apply safe and unsafe tests defined in the previous section on the $\otimes$-composition defined as below.

Definition 8 ($\otimes$-Composition of System Components Moving Synchronously). Given n compatible synchronized sequential ASMs $M'_1, \ldots, M'_n$, the $\otimes$-composition of $M'_1, \ldots, M'_n$ is the multi-agent ASM $M = \otimes_{i=1}^{n} M'_i$ with synchronized agents $a_1, \ldots, a_n$, constructed as follows:

1. a finite indexed set of programs $\Pi'_1, \ldots, \Pi'_n$ (modules) named as $\nu_1, \ldots, \nu_n$;
2. the vocabulary $\Upsilon = \bigcup_{i=1}^{n} \Upsilon'_i \bigcup \{Mod, \nu_1, \ldots, \nu_n\}$, the function *Mod* assigns to each agent its program name; dynamic function symbols in Υ are classified as follows:
   - 2a. $F_{Sh}(M) = \bigcup_{i,j=1}^{n, i \neq j} (\{f | f \in F_{Mon}(M'_i) \bigcap F_{Out}(M'_j)\}) \bigcup \bigcup_{i=1}^{n} F_{Sh}(M'_i)$
   - 2b. $F_{Mon}(M) = \bigcup_{i=1}^{n} F_{Mon}(M'_i) \setminus F_{Sh}(M)$
   - 2c. $F_{Out}(M) = \bigcup_{i=1}^{n} F_{Out}(M'_i) \setminus F_{Sh}(M)$

2d. $F_{Ctrl}(M) = \bigcup_{i=1}^{n} F_{Ctrl}(M_i')$
where $F_{Sh}(M)$ is constituted by all the functions monitored for some components and of output for others and by all the functions which are shared in some components;

3. a collection of Υ-structures being the *states* of M which are constructed from the $\mathcal{S}(M_i')$ in this way: for every n-uple $X_1', \ldots, X_n'$ of states of $M_1', \ldots, M_n'$ such that all the shared locations are identically evaluated, construct a Υ-structure X which interprets symbols in Υ as in $X_1', \ldots, X_n'$;
4. a collection of Υ-structures being the *initial states* of M constructed from $\mathcal{I}(M_i)$ as in step 3.

5.1 Case Study: Two Interacting Synchronized Telephones

Caller and Callee specifications may also be composed as two interacting devices with operation $\otimes$. At first, we parameterize Caller and Callee with *Self* and correctly refine them. Then we substitute in both programs the function symbol *Self* with the agent names *cr* and *ce* by obtaining $\text{Caller}' = \text{Caller}[Self \leftarrow cr]$ and $\text{Callee}' = \text{Callee}[Self \leftarrow ce]$.

The compatibility of Caller′ and Callee′ is trivially verified. The function test is not satisfied since $F_{Ctrl}(\text{Caller}') \bigcap F_{Ctrl}(\text{Callee}') = \{mode\}$ and $F_{Sh}(\text{Caller}') \bigcap F_{Sh}(\text{Callee}') = \{calledby\}$. Instead of going on with the application of the safe test, let us do the following consideration: although *mode* is shared between Caller′ and Callee′, it is parameterized with *cr* in Caller′ and with *ce* in Callee′. That implies that inconsistent updates for *mode* will never occur in the compound specification. In cases simple like this, an inspection of the specifications allows us to answer about the consistency of the system without going on in the application of further tests. In more complex cases, this could not be possible and we would end up with the application of the unsafe test. Notice that the unsafe test can be successfully applied in our example.

Applying the composition operation we obtain a multi-agent ASM with synchronous agents which is consistent with respect to updates.

5.2 Case Study: The Railroad Crossing Problem

The two components of the railroad crossing specified in [12], namely the `gate` and the `controller`, can be consistently composed as two interacting devices. At first, we parameterize rules of `gate` and `controller` with *Self*. Then, we substitute in both programs the function symbol *Self* with the agents names *gt* and *ctrl* by obtaining $\texttt{gate}' = \texttt{gate}[Self \leftarrow gt]$ and $\texttt{controller}' = \texttt{controller}[Self \leftarrow ctrl]$. It is easy to see that the compatibility condition is satisfied by the components and the consistency of the system with respect to updates is trivially verified since there are no functions updated in both components.

With the application of the $\otimes$-composition, we obtain a consistent multi-agent ASM with synchronous agents.

6 Composition of Distinct System Components Moving Asynchronously

In our third case, we take as input n sequential ASMs $M_1, \ldots, M_n$ which describe the behavior of components of a system which move asynchronously. The M_i are defined as in the previous section and, in order to be composed, we take their modified version M_i'.

The composition of asynchronous components open the door to further main issues like dependencies and interferences between the moves of the components, local parallelism and synchronism. Although such arguments are very interesting and applicable in practice, we address them as future work and focus on the definition of a composition operation consistent with respect to simultaneous updates.

Compatibility and Consistencies Tests. The compatibility condition is identical to that one in the previous section, it must be verified before the composition operation and a negative result prevents from integrating the components. Consistency tests are the same as in the previous section and are applied in the same way. The definition of consistency with respect to updates for multi-agent ASMs with asynchronous agents – under the assumption that every agent makes a move as soon as it is enabled – is the following:

Definition 9. *A multi-agent ASM M with asynchronous agents is* consistent *with respect to updates if it cannot happen that from a state X of M, moves x and y from agents a and b are independently fired and x and y respectively originate updates $(f, (\bar{a}), b_1)$ and $(f, (\bar{a}), b_2)$ such that $b_1 \neq b_2$, with $f \in \Upsilon$ vocabulary of M, $\bar{a}, b_1, b_2$ elements of X.*

Definition 10 ($\otimes$-Composition of System Components Moving Asynchronously). Given n compatible not synchronized sequential ASMs $M_1', \ldots, M_n'$, the $\otimes$-composition of $M_1', \ldots, M_n'$ is the multi-agent ASM $M = \otimes_{i=1}^{n} M_i'$ with asynchronous agents $a_1, \ldots, a_n$ constructed as follows:

1. a finite indexed set of programs $\Pi_1', \ldots, \Pi_n'$ (modules) named $\nu_1, \ldots, \nu_n$;
2. the vocabulary $\Upsilon = \bigcup_{i=1}^{n} \Upsilon_i' \bigcup \{Mod, \nu_1, \ldots, \nu_n\}$, *Mod* assigns to each agent its program name; Υ has the same dynamic function classification as in the synchronous case;
3. a collection of Υ-structures being the states of M which are constructed from the $\mathcal{S}(M_i')$ as in the synchronous case;
4. a collection of Υ-structures being the initial states of M constructed from $\mathcal{I}(M_i)$.

6.1 Case Study: The Production Cell

The production cell case study is useful to test the applicability of the composition operation $\otimes$ for asynchronous components and to discover some interesting

features of it. One of them is the possibility to construct the entire system by first composing components which are closely related to each other on the base of their local properties, and subsequently combining the partial composition into the system, checking its consistency and its main properties. For the production cell, the partial composition of the components FeedBelt, ElevatingRotaryTable and Robot1 (the first one of four automata describing the behavior of the Robot) illustrates that. Once the $\otimes$-composition of these components is constructed resulting in a consistent multi-agent ASM with asynchronous agents FB, ERT and R1 (details are skipped for space reasons), it is possible to focus on formulating and proving the properties of such piece of the system as shown in the following section.

7 Incidence of Composition Operations on Proof of System Properties

The composition operations introduced in this paper guarantee that the compound system is consistent with respect to updates. Nevertheless, inconsistencies related to the behavior of the system may still be present. In fact, due to the composition task, the components may lose or gain some behaviors and the system itself may not meet properties expected to hold. In this section, we define the property of *behavioral correctness of a system with respect to its components* and show how to define, analyze and prove partial or global properties of a system.

7.1 Behavioral Correctness of a Compound System with Respect to Its Components

In order to guarantee that, after the integration, the behaviors of the components must be preserved in the compound system. Therefore, the following property must hold:

Property. For every component of a compound system, the set of runs of the component's projection over the system coincides with the set of runs of the component.

Our composition operations do not guarantee the property above. Nevertheless, the $\otimes$-composition guarantees a partial version of the property, while the $\oplus$-composition gives a way to check whether the property is satisfied helping in localizing possible inconsistencies.

Behavioral Correctness of a $\otimes$-Composed System. When $\otimes$-composing a system, the correctness of the compound system in general is not guaranteed. A $\otimes$-compound system satisfies the following weaker property:

$\otimes$-Property. Given a system $M = \otimes_{i=1}^{n} M_i$, for every component M_i of M, the set of runs of the projection of M_i over M is included in the set of runs of M_i.

The proof follows from the definition of projection of a component over a $\otimes$-composed system given below:

Definition 11 (Projection of a Component over a $\otimes$-Composed System). *Given* $M = \otimes_{i=1}^{n} M_i$*, for every* $j \in \{1, \ldots, n\}$*, the* projection of M_j over M *is the sequential ASM* $\tilde{M}_j$ *having:*

- $\tilde{\Upsilon}_j = \Upsilon_j$, $\mathcal{S}(\tilde{M}_j) = \{View_{a_j}(X) | X \in \mathcal{S}(M)\}$,
- $\tilde{\Pi}_j = \Pi_j$,
- $\tau_{\tilde{M}_j}(X) \equiv_{Def} X + \Delta(\tilde{M}_j, X)$ *for every* $X \in \mathcal{S}(\tilde{M}_j)$.

Behavioral Correctness of a $\oplus$-Composed System. A $\oplus$-composed system is, in general, not correct with respect to its components. Nonetheless, it is possible to check whether every component and its projection over the system, constructed as shown in Definition 12, have identical behavior by comparing their programs as shown below.

Definition 12 (Projection of a Component over a $\oplus$-Composed System). *Given* $M = \oplus_{i=1}^{n} M_i$*, for every* $j \in \{1, \ldots, n\}$*, the* projection of M_j over M *is the sequential ASM* $\tilde{M}_j$ *having:*

- $\tilde{\Upsilon}_j = \Upsilon_j$, $\mathcal{I}(\tilde{M}_j) = \mathcal{I}(M_j)$,
- $\tilde{\Pi}_j$*, the subset of rules* R *of* Π *of vocabulary* $\tilde{\Upsilon}_j$,
- $\tau_{\tilde{M}_j}(X) \equiv_{Def} X + \Delta(\tilde{M}_j, X)$ *where* X *is a* $\tilde{\Upsilon}_j$*-structure,*
- $\mathcal{S}(\tilde{(M_j)})$*, the set of* $\tilde{\Upsilon}_j$*-structures individuated by* $\mathcal{I}(\tilde{M}_j)$ *and* $\tau_{\tilde{M}_j}$.

For every component M_i, check whether $\Pi_i = \tilde{\Pi}_i$. If they are equal, then M_i and $\tilde{M}_i$ have identical behaviors. If $\Pi_i \subset \tilde{\Pi}_i$, then there are some rules in $\tilde{\Pi}_i$ not in Π_i. These rules could make the projections to behave very differently from the components. Sometimes the additional rules don't bring any contribute and therefore M_i and $\tilde{M}_i$ have identical behaviors; sometimes they interfere with the already existing behaviors and therefore $runs(M_i) \neq runs(\tilde{M}_i)$; other times they may just add new behaviors without affecting the already existing ones and therefore, $runs(M_i) \subseteq runs(\tilde{M}_i)$. In all of the faulty situations considered above, the knowledge of the set of rules provoking the alteration in the component behavior, helps in fixing the problem giving a hint of what a possible solution may be.

Example: Caller $\oplus$ Callee. Computing the projection of the Callee over (Caller $\oplus$ Callee) and comparing it to the Callee component, we discover that the projection presents additional behaviors and that some properties are modified. The Caller specification is not subject to any behavioral change due to the composition.

7.2 Definition, Analysis and Proof of System Properties

The properties we consider concern the interactions between components, between components and environment, and between system and environment. We hide internal operations and introduce instruments to formulate and prove precise claims about what a system does. Definitions given below concern systems with synchronous or with asynchronous agents satisfying the constraint that, at each step, at most only one agent moves. Under this constraint, both categories of compound systems have runs which can be reduced to quasi-sequential runs.

Definition 13 (Input/Output Properties). *An* input/output property P *describes the behavior of an interface between the system and the environment. It consists of a set $f(P)$ of external functions, and of a set of (finite or infinite) sequences of first-order structures over $f(P)$ which is called* runs(P).

Definition 14 (Interaction Properties). *An* interaction property P *describes the behavior of an interface between components of the system and between components and the environment. It consists of a set $f(P)$ of external and interaction functions, and a set of (finite or infinite) sequences of first-order structures over $f(P)$ which is called* runs(P).

Definition 15 (Input/Output Runs). *Let ρ be a run of the ASM M with vocabulary Υ. The* input/output run ρ' *of ρ is obtained from ρ by replacing every state X_i, $i = 0, 1, \ldots$, with X_i', the reduct of X_i to $\Upsilon \setminus (F_{Sh}(M) \bigcup F_{Ctrl}(M))$, and identifying subsequent identical structures. The collection of input/output runs for M is called* externalRuns(M).

Definition 16 (Interaction Runs). *Let ρ be a run of the ASM M with vocabulary Υ. The* interaction run ρ' *of ρ is obtained from ρ by replacing every state X_i, $i = 0, 1, \ldots$, with X_i', the reduct of X_i to $\Upsilon \setminus F_{Ctrl}(M)$, and identifying subsequent identical structures. The collection of interaction runs for M is called* interactionRuns(M).

Definition 17. *An ASM M satisfies an input/output property P if $F_{Mon}(M) \bigcup F_{Out}(M) = f(P)$ and* externalRuns(M) $\subseteq$ runs(P).

M satisfies an interaction property Q if $F_{Mon}(M) \bigcup F_{Out}(M) \bigcup F_{Sh}(M) = f(Q)$ and interactionRuns(M) $\subseteq$ runs(Q).

As properties, we consider safety and liveness properties. Remind that safety properties claim that some particular bad things never happen, while liveness properties state that some particular good things eventually happens. These properties, when represented as suitably selected sequences of states, have particular structures which allow to recognize them as safety or liveness properties.

Definition 18 (Structure of a Safety Property). *A property P is a* safety property *if it satisfies the following conditions:*

- runs(P) *contains sequences with more than the initial state;*

- runs(P) *is* prefix closed*, that is, if* $\beta \in$ runs(P) *and* β' *is a finite prefix of* β*, then* $\beta' \in$ runs(P)*;*
- runs(P) *is* limit closed*, that is, if* $\beta_1, \beta_2, \ldots$ *is an infinite sequence of finite sequences in* runs(P)*, and for each* i*,* β_i *is a prefix of* β_{i+1}*, then the unique sequence* β *that is the limit of the* β_i *is also in* runs(P)*.*

Definition 19 (Structure of a Liveness Property). *A property* P *is a* liveness property *provided that every finite sequence over* $f(P)$ *structures have some extension in* runs(P)*.*

Now we give some examples of application showing how the definitions above help in proving that an ASM M satisfies a safety or liveness property.

Example 1. We want to show that the compound subsystem $M = FB \otimes ERT \otimes R1$ of Feed Belt, Elevating Rotary Table and Robot1, constructed in subsection 6.1 satisfies the **safety property** *"The Feed Belt does not put metal blanks on the table if the latter is already loaded or not stopped in loading position"* as defined in [3].

We can translate the safety property in an interaction property P such that:

- $f(P) = \{$*PieceInFeedBeltLightBarrier, UnLoadPosReached, LoadPosReached, FeedBeltFree, TableLoaded, TableInLoadPos, TableInUnloadPos, UnLoadingTableCompleted,UnloadPressPosReached*$\}$,
- the states are structures interpreting symbols in $f(P)$ as Booleans, and *runs*(P) are sequences ρ of states holding the following property: $\forall\ X_i \in \rho$ such that *PieceInFeedBeltLightBarrier* = **true** there is no X_{i+1} such that (*PieceInFeedBeltLightBarrier* = **false** $\wedge$ (*TableLoaded* = **true**$\vee$ *TableInLoadPos* = **false**).

It is easy to check that P is a safety property according to its structure and the definition 18. M satisfies P since $f(P) = F_{Out}(M) \bigcup F_{Mon}(M) \bigcup F_{Sh}(M)$ and *interactionRuns*(M) $\subseteq$ *runs*(P) (trivially proved by induction).

Example 2. Consider the **liveness property** *"Every blank inserted into the system (by initialization or via the Feed Belt) will eventually arrive at the end of the Deposit Belt (and then be dropped by the crane on the Feed belt again) and will have been forged"* of the production cell as in [3].

Since the activation (and deactivation) of photoelectric cells, sensors and switches witnesses the liveness of the system, we first prove the liveness of the components which we define as input/output properties, and the liveness of components relationships defined as internal interaction properties. An internal interaction property is a variant of an interaction property taking only shared functions of the partial composition of the interested components. Being all the properties verified over the specification, we can state the liveness of the system (proofs are skipped for lack of space).

8 Conclusion and Future Development

We presented two composition operations for sequential ASMs in order to respectively construct the behavioral specification of a device from its features and to build a faithful and polyhedral description of a system from its components without flatten them, preserving the interconnections between them and taking into account the different perspectives of the parties. Both operations are equipped with compatibility conditions for the partial specifications and with tests to check the consistency of the compound system with respect to updates. Further, it is shown that the operations can be used in order to prove partial or global properties of the system and to check its behavioral inconsistencies.

We are currently working on the definition and proof of properties regarding systems with asynchronous agents by getting rid of the constraint such that, at each step, at most one agent moves and by allowing the most general case of partially ordered runs. Further, we are focusing on problems of dependencies of moves from different agents in partially ordered runs and of partial synchronism of agents.

Up to now updates and behavioral inconsistencies have been manually detected. We plan to automatize this process by means of theorem provers (as the one developed in the Key-Project [1]) or model checkers.

References

[1] Ahrendt, W., Baar, T., Beckert, B., Giese, M., Haehnle, R., Menzel, W., Mostowsky, W., Schmitt, P. H.,(2002) *The KeY System: Integrating Object-Oriented Design and Formal Methods.* In Procs. of FASE at ETAPS 02. 339

[2] Blass A., Gurevich Y., (2001)*Abstract State Machines Capture Parallel Algorithms.* ACM Transactions on Computational Logic, to appear, MSR-TR-2001-117. 324, 328

[3] Boerger, E., Mearelli, L., (1997) *Integrating ASMs into the software Development Life Cycle*, JUCS vol. 3, n. 5, pp. 603-665. 324, 328, 331, 338

[4] Boerger, E., Riccobene, E., Schmid, J., (2000) *Capturing Requirements by Abstract State Machines: The Light Control Case Study.* JUCS, vol. 6, n. 7, pp. 597-620. 329

[5] Boerger, E., (2002) *The Origins and Development of the ASM Method for high level System Design and Analysis*, JUCS vol. 8, n. 1, pp. 2-74. 324

[6] Boerger, E., Schmid, J., (2000) *Composition submachines concepts for sequential ASMs.* In P. Clote and H. Schwichtenberg, eds. LNCS 1862, pp. 41-60. 325

[7] Easterbrook, S., Nuseibeh, B., (1996) *Using ViewPoints for Inconsistency Management.* BCS/IEE Software Engineering Journal, pp. 31-43, January 1996. 324, 327

[8] Easterbrook, S., Chechik, M., (2000) *A Framework for Multi-Valued Reasoning over Inconsistent ViewPoints*, In Procs. ICSE 2001.

[9] Finkelstein, A. C. W., Goedicke, M., Kramer, J., and Niskier, C., (1989) *Viewpoint Oriented Software development: Methods and Viewpoints in Requirements Engineering*, in Algebraic Methods II: theory, tools and applications, Bergstra, J. A. & Feijs, L. M. G. eds (Springer Verlag) 1991, LNCS 490, 29-54. 327

[10] Gurevich, Y., (1995) *Evolving Algebras 1993: Lipari Guide*, in E. Boerger, editor, Specification and Validation Methods, pp. 9-36. Oxford University Press, 1995. 324, 325, 326

[11] Gurevich, Y., (2000) *Sequential Abstract State Machines capture sequential algorithms*, in ACM Transaction on Computational Logic Volume 1, Number 1. 324, 325, 326, 329, 331

[12] Gurevich, Y., (1996) *The Railroad Crossing Problem: An Experiment with instantaneous Actions and Immediate Reactions*, Proc. CSL'95. LNCS 1092, pp. 266-290. 324, 328, 333

[13] Lewerentz, C., Linder, T. editors (1995) *Formal Development of Reactive Systems. A Case Study "Production Cell"*. Number 891 in LNCS. 328

[14] Linder, T. (1995). *Task description.* (891):9-21. 328

[15] Staerk, R. F., Nanchen, S., (2001) *A logic for Abstract State Machines*, JUCS, 7(11):981-1006. 324

Deciding the Verification Problem for Abstract State Machines

Antje Nowack

Mathematische Grundlagen der Informatik
RWTH Aachen, D-52056 Aachen, Germany
nowack@informatik.rwth-aachen.de

Abstract. We study the decidability of general verification problems for ASMs. This corresponds to the question whether all computations of an ASM satisfy a property (usually expressed in some temporal logic). Because of undecidability in the general case, one has to identify classes of ASMs and classes of properties in order to receive positive decidability results. One idea for this could be to put simple restrictions on the ASMs as forbidding certain advanced constructs. But it turns out that this does not help much. Additionally, the restrictions in those cases are very strong. Consequently, one has to look for others. Such restrictions may result from techniques which are intended to be used in the proofs of decidability. One proof technique is the reduction to a decidable reasoning problem in logic. In [8] and [10], this has been applied to receive some positive decidability results.
In this paper, we identify further classes of ASMs and classes of properties for which the general verification problem is decidable. Each proof is a reduction to the satisfiability problem of a fragment of linear temporal first-order logic which has respectively been proved to be decidable in [6] or [7].

1 Introduction

Abstract State Machines. Abstract State Machines (ASMs) provide a formal method for transparent design and specification of complex dynamic systems. They combine advantages of informal methods (understandability, executability) with the advantages of formal methods (precision and applicability of mathematical methods and results). The executability of ASM-specifications yields the possibility to find already a part of the errors made in a specification via testing.

However, the ASM-method is a formal method. There is a well defined semantics and this allows the application of mathematical methods to analyze ASMs. In particular, these are often results from mathematical logic as ASMs are based on first-order structures.

The Verification Problem. The aim of verifying ASMs is to prove that the behaviour of an ASM specification coincides with the desired one. In contrast to testing, one covers *all* possible cases and not only a (finite) part.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 341–355, 2003.
© Springer-Verlag Berlin Heidelberg 2003

It would be the ideal case to have an algorithm deciding, for a given ASM and a given property (typically formulated in some temporal logic), whether the ASM satisfies the property.

There are mainly two possible meanings for the question whether "the behaviour of an ASM specification coincides with the desired one":

1. Do all computations of an ASM satisfy a property? (general verification problem)
2. Does the computation of an ASM with a given initial structure satisfy a property? (restricted verification problem)

Model checking is an algorithmic method for checking whether a system satisfies a specification. The system is modeled by a structure (e.g. a transition system) and the specification is a formula in an appropriate logic. Then one checks whether the structure is a model of the formula.

Meanwhile, model checking has become an established industrial practice. Model checking algorithms for finite state systems proceed by a systematic examination of the entire state space. In most cases, it is not possible to exploit this approach for the verification of ASMs as one deals with infinite state spaces. Consequently, one has to look for other ways to carry out model checking.

Before treating the question whether an ASM satisfies a property, one has to consider the question, how far the (restricted or general) verification problem for ASMs is decidable.

Outline. A first idea for receiving decidability results would be to impose simple restrictions on ASMs and logics as forbidding certain advanced constructs. But this leads to positive decidability results only in a very few and rather weak cases. In order to receive further positive results, one has to consider other kinds of restrictions which often do not seem to be very intuitive. The reason for this may be that one does not start with a class of ASMs and a class of properties but with the intention to find some restrictions fitting to an idea for a proof. One possibility for a proof is a reduction to a decidable reasoning problem in logic. If, in this case, we start from the reasoning problem then we may receive appropriate restrictions on ASMs and properties. In [8] and [10], some positive decidability results concerning the general verification problem have been presented. The proofs of those results are reductions to the (finite) satisfiability problem of a logic which is decidable.

In this paper we will also apply this technique to identify further classes of ASMs and fragments of temporal first-order logic for which the general verification problem is decidable. The fragments used in the reductions are fragments of linear temporal first-order logic. The decidability of the satisfiability problems of these fragments has been proved in [6] and [7].

2 Simple Restrictions and Survey of Known Decidability Results

As this paper is written for the ASM community, we assume that the reader is familiar with ASMs. For a detailed introduction to ASMs see e.g. [5].

If no restriction is given then the guard of an if-clause, forall-rule or choice-rule may be any first-order formula (possibly containing quantifiers).

2.1 Simple Restrictions on ASMs

One of the ideas for identifying verifiable classes of ASMs and formalisms could be to forbid certain advanced construction elements or restricting their use in a rather simple way. Advanced construction elements are those which allow rules of the form **forall** $\bar{x} : \varphi(\bar{x})$ **do** Π **endforall**, the choice of an element from the domain of a state or the use of import.

It turns out that the restricted verification problem for full branching time first-order logic restricted to vocabularies containing only constant and relation symbols and the class $\mathcal{K}_0$ of ASMs is decidable. The class $\mathcal{K}_0$ contains all ASMs over vocabularies containing only constant and relation symbols without choice of an element from the domain of a state and without rules of the form **forall** $\bar{x} : \varphi(\bar{x})$ **do** Π **endforall** (but the use of import and the choice of one rule from two are allowed).

Decidability for the general verification problem is received by e.g. restricting the guards of if-clauses to the guarded fragment of first-order logic which is defined in Subsection 3.4.

If one restricts the specification logic to the first-order variant of Hennessy-Milner logic (in addition to first-order constructs, we only allow formulae of the form $\Diamond\varphi$ expressing that there exists a successor satisfying φ) then no restriction on the ASMs is necessary in the case of the restricted verification problem. Already a restriction similar to the guarded fragment of first-order logic leads to decidability in the case of the general verification problem (where the use of forall corresponds to universal quantification and choice to existential quantification). Note that the restriction similar to the guarded fragment contains a restriction to vocabularies containing only constant and relation symbols.

Weaker restrictions on ASMs or the specifying logic that we have received in this way do only lead to undecidability results.

From these considerations, one can see that such simple restrictions do not lead to decidability results in many cases.

The first-order variant of Hennessy-Milner logic (HM-FO) is a rather weak logic. A given formula only allows propositions about a fixed number of steps of an ASM (bounded by the maximal number of nested $\Diamond$). However, there is no uniform bound for all formulae. Therefore, for some applications, HM-FO may already be strong enough.

On the other hand, the class $\mathcal{K}_0$ of ASMs does not provide too many possibilities. The reason is that it is not possible to update the content of locations with

elements which are not from the reserve (addressable via import) and which are not the content of a constant. Therefore, all locations of the input state containing at least one such element are static.

For receiving further decidability results, one has to consider classes that result from other kinds of restrictions than those above. Such restrictions have already been considered in [8] and [10]. They are briefly resumed in the rest of this section.

2.2 Nullary Programs

In [8], the class of nullary (ASM-)programs is defined. A nullary program is a nondeterministic basic ASM-program where every dynamic function is nullary. "Basic" means the following in this context:

- the guards in the if-clauses and the choose-rules are not allowed to contain quantifiers
- the use of import or forall is not allowed

M. Spielmann has proven that the general verification problem for the existential and the universal fragment of the fragment of a CTL*-like logic containing only state formulae is decidable. The proof is done via a reduction to the satisfiability problem of existential transitive closure logic.

2.3 ASM-Transducers

In [10], ASM-transducers have been defined. A state of a transducer is a relational database containing the (dynamic) memory for the (static) database of the transducer. In every computation step, the transducer receives from the environment all input relations and reacts by producing output relations and by updating its memory. The memory contains all dynamic data of the transducer necessary to keep track of ongoing transactions.

An ASM-transducer program is a finite set of rules of the form

$$\textbf{if } \varphi \textbf{ then } (\neg)R(\bar{x})$$

where φ is an FO-formula and R is a memory-relation or an output-relation. $R(\bar{x})$ occurs positively if R is an output relation. If a location with an output-relation is not updated by the program then its content is false in the next step. Free variables are considered as bound by forall. In the case that $R(\bar{x})$ occurs negatively in the above rule, the content all locations $(R, \bar{a})$ such that $\varphi[\bar{x}/\bar{a}]$ is satisfied is set to false. If it occurs positively then the content of these locations is set to true. Initially, all memory relations are empty.
In [10], the following restrictions have been introduced:

1. The database is provided and fixed during verification.
2. The maximal input flow, to which an ASM transducer is exposed, is a priori bounded.

3. The arities of the relations are bounded a priori.
4. The specification logic does not admit existential quantification into temporal contexts, only universal (this fragment is called UT).

The general verification problem for ASM-transducers and the logic UT is decidable in the following cases (see [10]):

- The first and the third restriction are satisfied.
- The first and the second restriction are satisfied.
- The second restriction is satisfied and only formulae from the fragment of UT are allowed where every quantifier of a formula is relativized by an atomic input fact.

3 Foundations

3.1 ASMs

Starting from here, all ASMs referred to in this paper are deterministic and satisfy the following conditions:

- the vocabulary of an ASM contains only relation and constant symbols
- import is not used
- the states of an ASM are finite (as the use of import is not allowed, it is not necessary to take the reserve into account)
- the guard of an if-clause or a forall-rule may be any first-order formula possibly containing quantifiers
- updates of the form $c := t$ where c is a constant symbol are not allowed

3.2 Linear Temporal First-Order Logic

Linear temporal first-order logic (LTFO) is obtained by adding to first-order logic FO the following rules for constructing temporal formulae:

If φ and ψ are formulae from LTFO then $\varphi \mathrm{U} \psi$ and $\varphi \mathrm{S} \psi$ are in LTFO.

The semantics for LTFO is the usual one for linear temporal first-order logic (see [6]):
LTFO is interpreted in structures of the form $\mathcal{M} = (\mathcal{F}, D, I)$ where $\mathcal{F} = (W, <)$ is a strict linear order, D is a non-empty set, the domain of $\mathcal{M}$, and I is a function associating with every moment of time $w \in W$ a first-order structure

$$I(w) = (D, P_0^{I(w)}, ..., c_0^{I(w)}, ...),$$

the state at time w, in which $P_i^{I(w)}$, for each i, is a predicate on D of the same arity as P_i and $c_i^{I(w)}$ is an element of D. We require that the interpretations of constants c_i do not change, i.e. $c_i^{I(w)} = c_i^{I(v)}$ for all $w, v \in W$.
An assignment in D is a function a from the set of variables to D. The truth-relation $(\mathcal{M}, w) \models^a \varphi$ in the model $\mathcal{M}$ under the assignment a is defined as follows:

- $(\mathcal{M}, w) \models^a P_i(y_1, ..., y_l)$ iff $P_i^{I(w)}(a(y_1), ..., a(y_l))$ is true in $I(w)$
- $(\mathcal{M}, w) \models^a \varphi \mathrm{S} \psi$ iff there is $v < w$ such that $(\mathcal{M}, v) \models^a \psi$ and $(\mathcal{M}, u) \models^a \varphi$ for every u in the interval $(v, w) = \{u \in W : v < u < w\}$
- $(\mathcal{M}, w) \models^a \varphi \mathrm{U} \psi$ iff there is $v > w$ such that $(\mathcal{M}, v) \models^a \psi$ and $(\mathcal{M}, u) \models^a \varphi$ for every $u \in (w, v)$

The semantics of propositional connectives and quantifiers is as usual.

In this paper, we consider discrete time. Therefore, a run $\rho = (A_i)_{i \in \mathbb{N}}$ of a deterministic ASM is a structure $\mathcal{M}_\rho = ((\mathbb{N}, <), D_\rho, I_\rho)$ where D_ρ is the domain of A_0 and $I_\rho(i) = A_i$ for all $i \in \mathbb{N}$. For real-time systems with continuous time, consult e.g. [2].

Intuitively, for a run ρ of an ASM on a structure A_0, $\mathcal{M}_\rho \models \varphi \mathrm{U} \psi$ means that there is a state B in the run with $B \models \psi$ and until this state is reached, φ holds in every state. It is important that a state is eventually reached in which ψ holds.

Abbreviations. We make use of the common abbreviations of temporal logic:

$$\begin{array}{ll} \mathrm{F}\varphi = \top \mathrm{U} \varphi & \text{(some time in the future)} \\ \mathrm{G}\varphi = \varphi \wedge \neg \mathrm{F} \neg \varphi & \text{(always in the future)} \\ \mathrm{X}\varphi = \bot \mathrm{U} \varphi & \text{(at the next moment)} \end{array}$$

3.3 Computation Graph and Verification Problems

Definition 1 The computation graph $C_\Pi(\mathcal{I})$ of an ASM-program Π and an input structure $\mathcal{I}$ is a temporal first-order model $(\mathcal{F}, D, I)$ where $\mathcal{F} = (W, R, s)$ is a transition system with domain W, transition relation R and pointed element s. $\mathcal{F}$ satisfies Ruv iff $I(v)$ is a successor of $I(u)$ with respect to Π.
D and I are defined analogously to the models for linear temporal logic.

In the case of a deterministic ASM, the transition system of collapses to the natural numbers with the usual successor relation and 0 as pointed element. The computation graph corresponds to a run.

The restricted verification problem (RVerify) and the general verification problem (GVerify) are defined formally, each for a class of ASMs and a logic.

Definition 2 Let $\mathcal{A}$ be a class of ASMs and $\mathcal{L}$ be a logic or a fragment of a logic. Then

- $\mathrm{RVerify}(\mathcal{A}, \mathcal{L}) = \{(\Pi, \mathcal{I}, \varphi) | \Pi \in \mathcal{A}, \varphi \in \mathcal{L}, C_\Pi(\mathcal{I}) \models \varphi\}$
  denotes the restricted verification problem for $\mathcal{A}$ and $\mathcal{L}$
- $\mathrm{GVerify}(\mathcal{A}, \mathcal{L}) = \{(\Pi, \varphi) | \Pi \in \mathcal{A}, \varphi \in \mathcal{L}, C_\Pi(\mathcal{I}) \models \varphi$ for all input structures $\mathcal{I}$ of $\Pi\}$
  denotes the general verification problem for $\mathcal{A}$ and $\mathcal{L}$

3.4 Guarded Fragment

The decidability of the general verification problem is strongly connected to the decidability of the (finite) satisfiability problem for certain logics.
The first reason is that many proofs are reductions to some (finite) satisfiability problem.

A further reason becomes clear if you consider the rather simple program Π = `if` φ `then res := true endif`. If φ does not contain `res` then every computation graph Π satisfies the formula (X res) iff $\neg\varphi$ is not finitely satisfiable.

In the following, we define the guarded fragment of first-order logic for which the (finite) satisfiability problem is decidable. It has been introduced in [1].

Definition 3 Let Υ be a vocabulary containing only constant and relation symbols. The guarded fragment GF of first-order logic is defined inductively as follows:

1. Every relational atomic formula (possibly an equation) belongs to GF.
2. GF is closed under propositional connectives $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$.
3. If $\psi(\bar{x}, \bar{y})$ is a formula in GF then $\exists\bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$ and $\forall\bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x}, \bar{y}))$ belong to GF provided that $\text{free}(\psi) \subseteq \text{free}(\alpha) = \bar{x} \cup \bar{y}$ and $\alpha(\bar{x}, \bar{y})$ is a relational atomic formula.

Here, free(ψ) means the set of all free variables of ψ.

Since the introduction of GF in [1], several guarded logics have been defined. They all have in common that every first-order quantifier is relativized by a guard. I.e., they have the form $\exists\bar{y}(\alpha(\bar{x}, \bar{y}) \wedge \psi(\bar{x}, \bar{y}))$ or $\forall\bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi(\bar{x}, \bar{y}))$, $\text{free}(\psi) \cup \{\bar{x}\} \subseteq \text{free}(\alpha)$. In GF, the guards are atomic formulae. As the translation of propositional modal logic to first-order logic (leading to the modal fragment) uses only guarded quantification, the modal fragment is a subset of GF. The guarded fragment generalizes the modal fragment as there is no restriction to the number of variables or the arities of the relations. Based on a number of results (see [1], [3]), Andréka, van Benthem, and Németi assume that the guarded nature of quantification is the main reason for the good model-theoretic and algorithmic properties of modal logic. However, most of the interesting properties are still expressible in GF. Generalizations of GF are the loosely guarded fragment LGF of FO (see [11]) and the clique-guarded fragment CGF of FO (see [4]). GF is strictly contained in CGF and every sentence in LGF is equivalent to a sentence in CGF.

Analogously to the guarded fragment of first-order logic, we can define the guarded fragment of temporal first-order logic GF(LTFO) where all first-order quantifiers are relativized by atomic formulae. Similarly, one can generalize e.g. the clique-guarded fragment of first-order logic to the clique-guarded fragment of temporal first-order logic CGF(LTFO).

4 Guarded ASMs

4.1 Guarded ASMs and the Specification Logic

Definition 4 A guarded ASM is a deterministic ASM satisfying the following conditions:

- update rules have the form $R\bar{c}_1 x \bar{c}_2 x ... \bar{c}_n := t$ where $R \in \Upsilon$ is a relation symbol (the lefthand-side of an update-rule contains at most one free variable)
- the guards of the if-clauses have to be from the guarded fragment of first-order logic
- forall-rules have the form **forall** $x : \varphi(\bar{x}, \bar{y})$ **do** Π **endforall** where φ is a relational atomic formula with $\text{free}(\varphi) \subseteq \text{free}(\Pi) \cup \{\bar{x}\}$

The class of all guarded ASMs is denoted by ASM_1^g.

Note that the restrictions on ASMs introduced in Subsection 3.1 are still valid.

In [6], Hodkinson et al. present fragments of linear temporal first-order logic for which satisfiability in $(\mathbb{N}, <)$ over finite, not expanding domains is decidable (abbreviation: $\text{FinSat}(\mathcal{L}, (\mathbb{N}, <))$). For this purpose, they first define the monodic fragment $\mathcal{TL}_1$ of linear temporal first-order logic.

$\mathcal{TL}_1 :=$ Set of all linear temporal first-order formulae φ such that any subformula of φ of the form $\psi_1 \text{U} \psi_2$ or $\psi_1 \text{S} \psi_2$ has at most one free variable

It is not allowed to quantify into temporal contexts with more than one variable.

Definition 5 The guarded fragment of $\mathcal{TL}_1$ is defined as follows:

$$\text{TGF}_1 := \mathcal{TL}_1 \cap \text{GF}(\text{LTFO})$$

In [7], it has been proved that $\text{FinSat}(\mathcal{TL}_1 \cap \text{CGF}(\text{LTFO}), (\mathbb{N}, <))$ is decidable. This implies the decidability of $\text{FinSat}(\text{TGF}_1, (\mathbb{N}, <))$.

4.2 Decidability of the General Verification Problem

In this subsection, we present the sketch of a proof for the decidability of $\text{GVerify}(\text{ASM}_1^g, \text{TGF}_1)$. It is a reduction to $\text{FinSat}(\text{TGF}_1, (\mathbb{N}, <))$.

Compared to full linear temporal logic, the restrictions on TGF_1 are rather strong but anyway, many interesting properties of ASMs are still expressible. For example, for safety-properties, the restriction of $\mathcal{TL}_1$ is no restriction. The reason is that in the case of safety-properties, there is an FO-sentence φ defining when a state is safe (a state A is safe iff $A \models \varphi$). A run is safe iff all of its states are safe. Therefore, a run is safe iff it satisfies $\text{G}\varphi$.

To simplify the argumentation, three definitions are given in advance.

Definition 6 A quasi-atom is a formula of the form $\alpha(\bar{x}) \wedge \bigwedge_{i=1}^{n} y_i^1 \neq y_i^2$ where α is an atomic formula with $\text{free}(\alpha) \subseteq \{\bar{x}\}$ and y_i^1, y_i^2 $(i \in \{1, ..., n\})$ are constant symbols or variables from $\bar{x}$.

Definition 7 An extended substitution is a finite set of simultaneous replacements of variables and constants by terms.

Definition 8 A quasi-atom $\beta(\bar{x}) = \alpha(\bar{x}) \wedge \bigwedge_{i=1}^{n} y_i^1 \neq y_i^2$ is unifiable with an atomic formula $\gamma(\bar{z})$ if α and γ are unifiable if there exists an extended substitution σ such that $\sigma(\alpha)$ and $\sigma(\gamma)$ are identical and $\sigma(\bigwedge_{i=1}^{n} y_i^1 \neq y_i^2)$ is satisfiable.

Lemma 1 *For every guarded ASM Π, there exists an effectively constructable sentence* $\text{consistent}_\Pi \in \text{GF}$ *such that for all states $\mathcal{A}$ of Π the following holds:*
$\mathcal{A} \models \text{consistent}_\Pi$ *iff the update set produced by Π on $\mathcal{A}$ is consistent.*

Proof. The construction of consistent_Π is straightforward. Two rules can only cause a collision if they update the same relation. Therefore, consider every atom appearing on the left-hand side of an update-rule. The formula can be directly gained from the respective guards.
For making clear the construction, consider the following guarded ASM rule Π:

do-in-parallel
  forall $x_1, x_2 : \gamma_1(x_1, x_2)$ **do**
    if $\alpha(x_1, x_2)$ then $Qx_1 :=$ true **endif**
  endforall
  forall $y_1, y_2, y_3 : \gamma_2(y_1, y_2, y_3)$ **do**
    if $\beta(y_1, y_2, y_3)$ **then** $Qy_1 :=$ false **endif**
  endif

Then

$$\text{consistent}_\Pi = \forall x_1 x_2 x_3(\gamma_2(x_1, x_2, x_3) \rightarrow (\beta(x_1, x_2, x_3) \rightarrow \neg\exists y(\gamma_1(x_1, y) \wedge \alpha(x_1, y))))$$

As the above rule is a guarded ASM rule, this formula is in GF. The construction obviously extends to arbitrary rules. □

Note that $\text{GF} \subseteq \text{TGF}_1$ and therefore, $\text{consistent}_\Pi \in \text{TGF}_1$.

Lemma 2 *For every guarded ASM Π there exist effectively constructable sentences $\psi_\Pi, \chi_\Pi \in \text{TGF}_1$ such that for all states $\mathcal{A}, \mathcal{B}$ of Π and all temporal structures $\mathcal{M} = ((\mathbb{N}, <), D, I)$ with $I(1) = \mathcal{A}$, $I(2) = \mathcal{B}$ and the update set produced by Π on $\mathcal{A}$ is consistent, the following holds:*

$\mathcal{A} \vdash_\Pi \mathcal{B}$ *iff* $(\mathcal{M}, 1) \models \psi_\Pi \wedge \chi_\Pi$ *and for all quasi-atoms $\alpha(\bar{x})$ not unifiable with the lefthand-side of an update-rule in Π $\alpha(\mathcal{A}) = \alpha(\mathcal{B})$*

Proof. First, the formula ψ_Π is constructed. It formulates the changes induced by Π and is defined by induction on Π:

- If Π has the form $(s := t)$ then $\psi_\Pi = (\mathrm{X}s) \leftrightarrow t$.
- If Π has the form **if** g **then** Π' **endif** then $\psi_\Pi = g \rightarrow \psi_{\Pi'}$.
- If Π has the form **do-in-parallel** Π_0 Π_1 then $\psi_\Pi = \psi_{\Pi_0} \wedge \psi_{\Pi_1}$.
- If Π has the form **forall** $\bar{x}$: $\alpha(\bar{x}, \bar{y})$ **do** $\Pi'(\bar{x}, \bar{y})$ **endforall** then $\psi_\Pi = \forall \bar{x}(\alpha(\bar{x}, \bar{y}) \rightarrow \psi_{\Pi'}(\bar{x}, \bar{y}))$.

ψ_Π is a sentence because Π has no free variables. In ψ_Π, the only temporal operators are those X which are introduced via an update rule. Therefore, subformulae of ψ_Π with a temporal operator outermost are of the form Xs where s is the left-hand side of an update rule in Π. By the definition of guarded ASMs, s and Xs contain at most one variable. Because of the restriction on the guards of Π, $\psi_\Pi \in \mathrm{TGF}_1$.

χ_Π avoids changes in those atoms appearing on the lefthand-side of an update-rule in Π not indicated by the program. □

If the update set produced by a program on a structure is not consistent then nothing changes:

Lemma 3 *For every guarded ASM Π there exists an effectively constructable sentence* $\mathrm{noChanges}_\Pi \in \mathrm{TGF}_1$ *such that for all states $\mathcal{A}, \mathcal{B}$ of Π and for all temporal structures $\mathcal{M} = ((\mathbb{N}, <), D, I)$ with $I(1) = \mathcal{A}$ and $I(2) = \mathcal{B}$, the following holds:*

$(\mathcal{M}, 1) \models \mathrm{noChanges}_\Pi$ *iff for all atoms $\alpha(\bar{x})$ appearing on the lefthand-side of an update-rule in Π $\alpha(\mathcal{A}) = \alpha(\mathcal{B})$*

Theorem 1 $\mathrm{GVerify}(\mathrm{ASM}_1^g, \mathrm{TGF}_1)$ *is decidable.*

Proof. The decidability of $\mathrm{GVerify}(\mathrm{ASM}_1^g, \mathrm{TGF}_1)$ is proved via a reduction to $\mathrm{FinSat}(\mathrm{TGF}_1, (\mathbb{N}, <))$.
Let Π be a guarded ASM program and φ be a TGF_1-formula over the same vocabulary as Π. A formula Φ_Π will be constructed such that

$$(\Pi, \varphi) \in \mathrm{GVerify}(\mathrm{ASM}_1^g, \mathrm{TGF}_1) \text{ iff } \mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi) \notin \mathrm{FinSat}(\mathrm{TGF}_1, (\mathbb{N}, <))$$

where trans_Π transforms a formula in TGF_1 into another formula in TGF_1 with respect to Π as follows. $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi) \in \mathrm{FinSat}(\mathrm{TGF}_1, (\mathbb{N}, <))$ iff there is a model $\mathcal{M} = ((\mathbb{N}, <), D, I)$ of $\Phi_\Pi \wedge \neg\varphi$ such that for all quasi-atoms $\alpha(\bar{x})$ not unifiable with the lefthand-side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$.
Note that this does *not* imply that for every model $\mathcal{M} = ((\mathbb{N}, <), D, I)$ of $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$ and for all quasi-atoms $\alpha(\bar{x})$ not unifiable with the lefthand-side of an update rule in Π, $\alpha(I(n)) = \alpha(I(m))$ for all $m, n \in \mathbb{N}$.
The formula $\Phi_\Pi \in \mathrm{TGF}_1$ has the form

$$\Phi_\Pi = \mathrm{G}((\mathrm{consistent}_\Pi \rightarrow (\psi_\Pi \wedge \chi_\Pi)) \wedge ((\neg\mathrm{consistent}_\Pi) \rightarrow \mathrm{noChanges}_\Pi))$$

where consistent$_\Pi$ is the GF-sentence from Lemma 1, ψ_Π and χ_Π are the sentences from Lemma 2 and noChanges$_\Pi$ is the one from Lemma 3.
Because $(\text{consistent}_\Pi \rightarrow (\psi_\Pi \wedge \chi_\Pi)) \wedge ((\neg\text{consistent}_\Pi) \rightarrow \text{noChanges}_\Pi)$ is a TGF$_1$-formula without free variables, Φ_Π is an TGF$_1$-formula. □

4.3 Other Verifiable Classes of ASMs

In [6], [7] and [12], the decidability of FinSat$(\mathcal{L}, (\mathbb{N}, <))$ has been proved for further fragments $\mathcal{L}$ of $\mathcal{TL}_1$. For many of these fragments $\mathcal{L}$, a class of ASMs can be defined for which the respective fragment (or to be more precise: the decidability of FinSat$(\mathcal{L}, (\mathbb{N}, <))$) implies the decidability of the general verification problem.

The results for the guarded fragment directly extend to the clique-guarded fragment and ASMs analogously defined to guarded ASMs but the condition for the guards in the if-clauses and the forall-rules are generalized according to the clique-guarded fragment. This leads to the defintion of clique-guarded ASMs.

A further example is the monadic fragment of $\mathcal{TL}_1$. Monadic ASMs are deterministic ASMs over vocabularies containing only constant symbols and *unary* relation symbols. Furthermore, the use of equality is not allowed.

The proof of the decidability of the corresponding verification problem can be done analogously to the one for guarded ASMs and it becomes even easier. The transformation of the constructed formula (as carried out in the proof for guarded ASMs) is not needed as one can avoid arbitrary changes by a formula linked conjunctively to the constructed one. Equality is only needed in a very restricted way. Namely, where the left-hand-side is a constant. Decidability of the satisfiability problem over finite, not expanding domains in this case can be proved in the same way as the one for satisfiability without equality (this could also be eliminated by disallowing the occurrence of constants in the lefthand-side of an update-rule).

4.4 Extension of Guarded ASMs by Interaction

The main difference between ASM transducers (see [10] or subsection 2.3) and guarded ASMs is that ASM transducers provide the possibility to interact with the environment. The other differences, like the static database part or the memory part of the transducers, are not really additional features.

Guarded ASMs can be extended by interaction quite easily. Instead of one input state (over the the vocabulary Υ of the program) for a run, there is a sequence of input states $\bar{I} = I_1 I_2 ...$ over the input vocabulary $\Upsilon_{in} \subseteq \Upsilon$. The notion of a run has to be modified as follows:
Let ρ be a sequence $(A_i)_{i\in\mathbb{N}}$ of states over Υ. ρ is the run of Π on $\bar{I}$ if

- $A_i|_{\Upsilon_{in}} = I_i$
- Let B_{i+1} be the successor of A_i with respect to Π. Then $A_{i+1}|_{\Upsilon-\Upsilon_{in}} = B_{i+1}|_{\Upsilon-\Upsilon_{in}}$ for all $i \in \mathbb{N}$.

The decidability result for guarded ASMs can be directly transferred to interactive guarded ASMs. Note that in the case of interactive ASMs, the general verification problem has to be slightly modified. Not only all input states have to be taken into account but all input states and sequences of states over the input vocabulary.

Analogously, one can extend the classes from subsection 4.3 by interaction.

4.5 Comparison to Known Decidability Results

Nullary ASMs. In subsection 2.2, the results on nullary programs, originally introduced in [8], have been shortly resumed.

The result on nullary programs and the results obtained in this paper are merely comparable. Nullary programs allow the update of location consisting only of a constant but no other location can be updated. No ASM-program except Skip is a guarded ASM-program and a nullary program. The updates in nullary ASMs are forbidden in guarded ASMs.

ASM-Transducers. On the first view, one may have the impression that the results from [10] are stronger than those presented in this paper. The reason for this impression could be the restrictions on the use of variables in interactive clique-guarded ASMs. But this impression is wrong.

The first and obvious point is that the verifiable properties are not the same. For ASM-transducers we can verify properties expressible in UT, for interactive clique-guarded ASMs we can verify properties expressible in $\mathrm{TCGF}_1 := \mathcal{TL}_1 \cap \mathrm{CGF}(\mathrm{LTFO})$. The two fragments do not coincide and for each of the two fragments there are properties expressible in the one but not in the other.

But this is not the essential difference. For demonstrating this, we consider an example. Using interactive clique-guarded ASMs, we want to model a very simple system governing the student data of a university.

At the end of his studies, a student has to take part in the exams of at least r combinable lectures. Only if he passes all exams then he receives a certificate. If he fails then he has the possibility to do r combinable exams again. If this was already his second try then he has to leave the university without a certificate. After every term, the results of the exams are submitted to the system and the current certificates and aborts are emitted. Internally, students are identified via their registration number. This allocation is static. Output data like aborts or certificates does not contain registration numbers.

In the following, we give an ASM-transducer and an interactive clique-guarded ASM which are equivalent and both try to model the above described system.

ASM-transducer:

relations

input : fail/2, pass/2
database : number/2, combinable/r
memory : try1/1, try2/1
output : certificate/1, abort/1
log : fail/2, pass/2,
certificate/1, abort/1

memory rules

if exam(n) **then** try1(n)

if exam(n) $\wedge$ try1(n) **then** try2(n)

output rules

if number$(s, n) \wedge (\exists l$ fail$(n, l)) \wedge$ try1(n) **then** abort(s)

if number$(s, n) \wedge \neg$try2(n) $\wedge \neg(\exists l$ fail$(n, l)) \wedge$
$(\exists l_1 ... l_r$ combinable$(l_1, ..., l_r) \wedge \bigwedge_{i \in \{1,...,r\}}$ pass$(n, l)) \wedge$ **then**
certificate(s)

The above ASM-transducer can be rewritten to an equivalent interactive clique-guarded ASM with the same input vocabulary. For better readability, do-in-parallel is left away and the rules are split into memory rules and output rules.

memory rules

forall n : true **do**
if exam(n) **then** try1(n) := true **endif**
if exam(n) $\wedge$ try1(n) **then** try2(n) := true **endif**
endforall

output rules

forall s, n : number(s, n) **do**
if try1(n) $\wedge \exists l$ fail(n, l) **then**
abort(s) := true
else abort(s) := false
endif
if $\neg$try2(n) $\wedge \neg(\exists l$ fail$(n, l)) \wedge$
$(\exists l_1 ... l_r$ combinable$(l_1, ..., l_r) \wedge$
$\bigwedge_{i \in \{1,...,r\}}$ pass$(n, l))$ **then**
certificate(s) := true
else certificate(s) := false
endif
endforall

forall s : true **do**
if $\neg \exists n$ number(s, n) **then**
certificate(s) := false
abort(s) := false
endif
endforall

where exam(n) abbreviates the formula $(\exists l$ pass$(n, l)) \vee (\exists l$ fail$(n, l))$.

With the results presented in this paper, we can verify any property expressible as a TCGF_1-formula without any further restrictions. The results in [10] do not yield this possibility for UT as we have to care for the restrictions introduced there. In our example, a fixed database would determine the number of students. A limited input flow would imply a limit on the number of carried out exams albeit whether passed or failed. Both restrictions are not suitable in the above case.

An example for a property expressible in TCGF_1 is that if you fail for the second time then you will never receive a certificate in the future. It can be formalized by the following TCGF_1-formula.

$$\mathrm{G}\ \forall n(\mathrm{fail}(n) \rightarrow \mathrm{X}\ \mathrm{G}\ \mathrm{try1}(n)) \wedge$$
$$\mathrm{G}\ \forall sn(\mathrm{number}(s,n) \rightarrow ((\mathrm{try1}(n) \wedge \mathrm{fail}(n)) \rightarrow \neg \mathrm{F}\ \mathrm{certificate}(s)))$$

Note, furthermore, that, in contrast to ASM-transducers the memory relations of an interactive clique-guarded ASM do not need to be initially empty but if you want to verify a property φ only for this case then you can do it by verifying $(\bigwedge_{R \in \Upsilon} \neg \exists x_1 ... x_{\mathrm{arity}(R)} R x_1 ... x_{\mathrm{arity}(R)}) \rightarrow \varphi$ instead of φ.

4.6 Complexity

[7] implicitly provides an algorithm deciding whether $\varphi \in \mathrm{FinSat}(\mathrm{TGF}_1, (\mathbb{N}, <))$ with deterministic time in $2^{2^{O(2^{|\varphi| \cdot \log |\varphi|})}}$.
The size of Φ_Π is linear in $|(\Pi, \varphi)|$ (the size of (Π, φ)). In the worst case, the size of $\mathrm{trans}_\Pi(\Phi_\Pi \wedge \neg\varphi)$ is in $O(2^{|(\Pi,\varphi)|})$.
The above implies that the worst-case time complexity of $\mathrm{GVerify}(\mathrm{ASM}_1^g, \mathrm{TGF}_1)$ is in $2^{2^{O(2^n \cdot 2^n)}}$.

The finite satisfiability problem for the guarded fragment of first-order logic is complete for 2EXPTIME (double exponential time). This directly implies that $\mathrm{GVerify}(\mathrm{ASM}_1^g, \mathrm{TGF}_1)$ is 2EXPTIME-hard.

The other fragments do not provide much better results concerning their worst-case complexity. E.g., the general verification problem for interactive monadic ASMs and the monadic fragment of $\mathcal{TL}_1$ is EXPSPACE-complete.

5 Conclusion

We have observed that the decidability results for both versions of the verification problem are tightly connected to decidability results in mathematical logic. This is the main reason for putting relatively strong restrictions on the ASMs and the specification logic in order to receive positive decidability results.

As the worst-case complexities of the satisfiability problem for expressive logics are usually quite high (if decidable), the worst-case complexities for both versions of the verification problem are also quite high.

To receive lower worst-case complexities, one has to accept a trade-off concerning the expressiveness. Then the question for applicability becomes more and more difficult to answer. But one should also take into account that these high complexities reflect only the worst-case. For applications, one would be more interested in the average-case complexity. This one strongly depends on the particular applications.

This is what one should have in mind when trying to identify further verifiable fragments.

Acknowledgements

I want to thank Erich Grädel for many helpful remarks on earlier drafts of this paper. I also want to thank the anonymous referees for their useful suggestions.

References

[1] H. Andréka, J. van Benthem, and I. Németi. Modal Languages and Bounded Fragments of Predicate Logic. Technical Report ILLC Research Report ML-96-03, ILLC, 1996. 347

[2] D. Beauquier and A. Slissenko. A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. *Annals of Pure and Applied Logic*, 113(1-3):13–52, 2002. 346

[3] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64:1719–1742, 1999. 347

[4] E. Grädel. Guarded fixed point logic and the monadic theory of trees. *Theoretical Computer Science*, 288:129–152, 2002. 347

[5] Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, EECS Dept, University of Michigan, 1997. 343

[6] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. *Annals of Pure and Applied Logic*, 106:85–134, 2000. 341, 342, 345, 348, 351

[7] I.Hodkinson. Monodic packed fragment with equality is decidable. Preprint, 2001. 341, 342, 348, 351, 354

[8] M. Spielmann. Automatic Verification of Abstract State Machines. In *Proceedings of 11th International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *LNCS*, pages 431–442. Springer-Verlag, 1999. 341, 342, 344, 352

[9] M. Spielmann. Model Checking Abstract State Machines and Beyond. In *International Workshop on Abstract State Machines ASM 2000*, LNCS, pages 323–340. Springer-Verlag, 2000.

[10] M. Spielmann. Verification of Relational Transducers for Electronic Commerce. In *19th ACM Symposium on Principles of Database Systems PODS 2000, Dallas*, pages 92–93. ACM Press, 2000. 341, 342, 344, 345, 351, 352, 354

[11] J. van Benthem. Dynamic Bits and Pieces. Technical Report LP-97-01, ILLC, University of Amsterdam, 1997. 347

[12] F. Wolter and M. Zakharyaschev. Axiomatizing the monodic fragment of first-order temporal logic, 2000. Submitted. 351

An ASM Semantics of UML Derived from the Meta-model and Incorporating Actions*

Ileana Ober

VERIMAG
Centre Équation - 2, avenue de Vignate, 38610 Gieres, France
ileana.ober@imag.fr

Abstract. We present an approach towards a formal dynamic semantics for UML using ASM. We aim to remain as close as possible to the standard definition of UML and to cover the operational part of the language with particular attention to the behavior description based on actions. To remain close to the standard UML, we automatically translate the UML metamodel in ASM. This allows to take into account all the concepts and relationships contained in the standard, and to minimize the changes subsequent to the frequent updates of the standard.
For the dynamic part, the particularity of our approach is that we focus on actions, as defined in our proposal to the OMG action semantics working group. We deal with concurrency, signal exchange, operation calls, general communication primitives, etc. We do not define the semantics of state machines, but we clearly define their place within the framework of our semantics. We also describe how the ASM domains and functions used in the semantics are built initially from a particular UML model.

1 Introduction

While UML is rapidly becoming the industry standard for modeling, its standard definition does not contain a precise semantics. The semantics of UML is defined, in Chapter 2 of [19], by a textual description coming with some meta-modeling description (an of abstract syntax of the language). A preliminary part of our research, not described in this paper, that can be found in [[15], [17]] consisted in defining a mechanism for behavior description based on actions. This work was done as a response to an official request by the OMG, in conjunction with the Action Semantics Working Group. To complete the behavior description mechanism based on actions, we formalize it in the framework of the entire UML language.

* The major part of this work was done while the author was at Telelogic (www.telelogic.com). Additional support provided by the OMEGA project IST-33522 (http://www-omega.imag.fr).

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 356-371, 2003.
© Springer-Verlag Berlin Heidelberg 2003

In this paper we present a UML formalization approach using Abstract State Machines (ASM) [10]. We have chosen the formalism of ASMs for its expressing power coupled with a high level of abstraction, that applied nicely to a related formalization targeting the SDL semantics [[6], [13]]. Our formalization approach can be regarded both in connection with the previous work, and as a stand-alone UML formalization approach. We use as starting point for our formalization the UML meta-model which we translate automatically in ASM. This allows us to consider all the concepts and relationships existing in the standard. Also, as the language is continuously subject to (in general minor) changes, our semantics can easily adapt to new versions. On the other hand, at the moment we write this paper a major update of the language is planned. Our approach will minimize the changes needed in the semantics.

For the dynamic part we use the actions for specifying behavior. Actions cover transformational and control-oriented behavior (through data access, assignments, loops and decision constructs), as well as interactive behavior (through remote operation calls, signal passing, etc). We describe the concurrency model. In this paper we give only the basic principles of the semantics definition and some examples. For more details on the actual semantics the reader is referred to the Annex B of [17].

The rest of the paper is organized as followings: in Section 2 we argue on the need for a precise semantics for UML and we highlight some points to take into account for the semantics definition. Section 3 gives some insight into the approach we have taken. We describe here the overall approach and the structure of the semantics, we briefly present the static and the dynamic parts of the semantics (§3.1-§3.2), and we end this Section by presenting how the semantics applies to specific models (§3.3). In Section 4 we present some related work and we try to position our research in this context. We end in Section 5 by drawing some conclusions and presenting future work directions.

2 Need for Precise Semantics of UML

The need for a precise semantics of UML was often stated [[7], [8]]. A sound semantics for UML allows to build tools that check models, simulate them, and generate code.

Currently, the UML semantics is informally defined in plain text and it is often unclear, ambiguous or it contains contradictory assertions. Tools containing compilers use various methods to solve these problems, which led to different actual UML semantics implemented by different UML tools.

Although the existence of an informal semantics does not preclude the fulfillment of these goals, the practice has proven that a formal semantics brings them closer. That is why we will use formal techniques for defining the UML semantics. Nevertheless, we do not see the definition of a formal semantics as a goal, it only represents a possible way towards increased precision and advanced tool support.

The definition of a precise semantics should take care of two essential UML characteristics: the fact that UML is intended to address the early phases of analysis and design, thus the need to handle incomplete and possibly inconsistent information, and the fact that UML is mostly seen as a family of languages [5], possibly

demanding contradictory semantics. The fact that UML shall be applicable on various domains with contradictory demands for the semantics of the same entity was often an argument against the definition of a precise semantics for UML.

This formalization approach is a continuation of previous efforts on defining a precise semantics to UML actions. These efforts [[17] chapter 7, [15]] resulted in the definition of a framework for behavior description based on actions.

UML state machines are incomplete without a precise actions semantics, as actions describe what happens on state and transitions. On the other hand, we can completely describe behavior using actions (and no state machine). Therefore by focusing on action we can treat complete specifications.

Our semantics applies to a domain where behavior is important and it is described through actions, thus we give special attention to behavior and communication. Part of the task of this approach is to make choices, when the standard leaves place for alternatives (e.g. we will use a specific model for concurrency, a certain semantics for signal exchange, etc.). This choices are of course arguable, but unavoidable to obtain a meaningful semantics.

We use Abstract State Machines [10] for our formalization because they allow an operational description at a high abstraction level. Subsequent changes to the semantics definition will demand a reasonable amount of extra-efforts and the high level of abstraction would solves some of the problems related to over-specification.

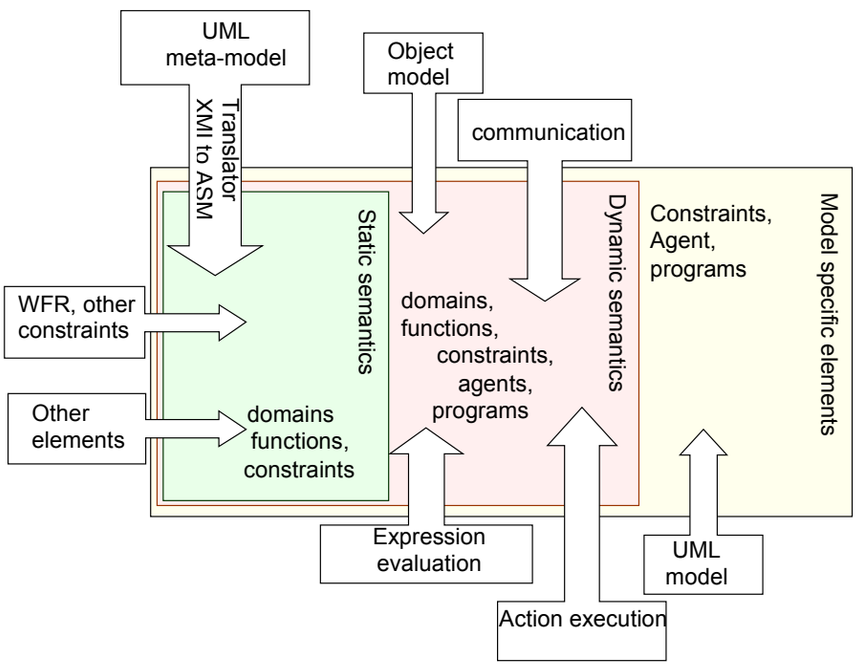


Fig. 1. Structure of the UML semantics

3 UML Semantics Definition

The definition of the ASM formal semantics of UML consists in associating to each UML model a particular multi-agent real-time ASM. Therefore, given an UML model, which is an instance of the UML meta-model, we have to identify its corresponding ASM agents, functions and domain names and for each agent the ASM program that describes its behavior.

Fig. 1 illustrates the structure of our formal semantics definition. As shown in this figure, the static semantics part expresses each UML basic concept in terms of ASM names and constraints. In the UML standard [19] the static semantics is defined using the UML meta-model, constrained through some integrity constraints, expressed as well-formedness rules (WFRs) described in OCL, or in plain text.

In order to best use the existing specification, as the UML meta-model contains information expressed in a semi-formal way, we have developed an automatic ASM generator that translates the information contained in the UML meta-model into ASM. In the next Section we will discuss this tool into more detail.

Our *static semantics* definition contains constraints corresponding to the WFRs stated in the definition of UML both in OCL and in plain text. Additionally, the static semantics offers access function to various elements of the static semantics, describes the method lookup mechanisms, etc.

The *dynamic semantics* is constituted by some ASM names, constraints and rules that describe the basis of behavior. These rules formalize the semantics of signal passing, of execution threads, of the signal queue etc. The dynamic semantics does not correspond directly to anything from the UML meta-model, although the basic concepts (operation, signal, etc) are part of the meta-model and have been translated into ASM in the static semantics part.

The dynamic semantics contains functions that describe the evaluation of expressions, and operational semantics of UML actions expressed in ASM, based on an execution engine that describes the basis of behavior (message passing semantics, time, etc.). When defining the dynamic semantics we often have to choose the semantics we give to various constructs, because the UML standard does not cover all details - either deliberately, or by omission.

As one can see from Fig. 1, the final result of the semantics definition is an ASM program equivalent to an arbitrary UML model. This ASM program may be used to generate code that will conform the semantics, or as input for ASM simulators which may execute symbolically the ASM program corresponding to an UML model. Some ASM tools offer already part of this functionality, some others may be further developed in order to offer a better coupling between the UML model specification and the simulations performed on the equivalent ASM program. Such tools could be made transparent to the UML modeler, which does not have to be aware of the fact that his model is internally translated into ASM, by feeding back at UML level the results of the simulation and verification.

3.1 Static Semantics

Currently, the static semantics of the standard UML is defined by the UML meta-model, by some informal textual descriptions and it contains constraints given as well

formedness rules [19]. Our static semantics definition has to capture all of these concepts in ASM. Additionally, the static semantics should contain ASM names for access functions and a method lookup mechanism.

Automatic Meta-model Translation Using XMItoASM

The first step of the UML semantics definition consists in giving the ASM names equivalent to UML concepts. The UML meta-model is a formalism for defining entities and relationships between them. In order to translate automatically the information contained in the UML meta-model into ASM, we have developed a tool that has as input the standard UML meta-model (in XMI form) and translates it into ASM. When applied on the UML meta-model, this translator, that we call *XMItoASM*, recovers names (domains and functions) and constraints from the UML meta-model classes, attributes, associations and generalizations.

Being a one-time only process, the translation of the UML meta-model could have been done by hand, but the automatic translation has the advantage that it minimizes the risk of translation errors and it allows an easy update of the ASM names and constraints in the event of further changes in the UML meta-model.

For exemplification we consider the excerpt from the UML meta-model (from the ***Core*** package) depicted in Fig. 2. The ASM names corresponding to the classes described in this example are contained at lines 1-9 in Fig. 3.

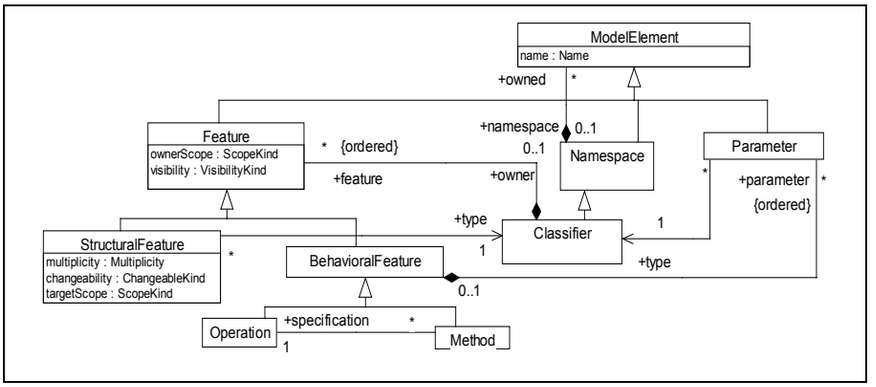


Fig. 2. UML meta-model extract

```
1  static domain UMLMODELELEMENT
2  static domain UMLFEATURE
3  static domain UMLNAMESPACE
4  static domain UMLPARAMETER
5  static domain UMLCLASSIFIER
6  static domain UMLSTRUCTURALFEATURE
7  static domain UMLBEHAVIORALFEATURE
8  static domain UMLOPERATION
9  static domain UMLMETHOD
10 constraint UMLNAMESPACE      ⊆ UMLMODELELEMENT
11 constraint UMLFEATURE        ⊆ UMLMODELELEMENT
12 constraint UMLPARAMETER      ⊆ UMLMODELELEMENT
```

```
13 constraint UmlBehavioralFeature      ⊆ UmlFeature
14 constraint UmlStructuralFeature      ⊆ UmlFeature
15 constraint UmlOperation      ⊆ UmlBehavioralFeature
16 constraint UmlMethod         ⊆ UmlBehavioralFeature
17 constraint UmlClassifier     ⊆ UmlNamespace
18 static ModelElement_name   :UmlModelElement→ UmlName
19 static Feature_ownerScope  :UmlFeature     → UmlScopeKind
20 static Feature_visibility  :UmlFeature     → UmlVisibilityKind
21 static StructuralFeature_multiplicity :UmlStructuralFeature →
                                  UmlMultiplicity
22 static StructuralFeature_changeability:UmlStructuralFeature→
                        UmlChangeableKind
23 static StructuralFeature_targetScope  :UmlStructuralFeature →
                                  UmlScopeKind
24 ModelElement_namespace     :UmlModelElement→ UmlNamespace-set
25 Namespace_owned      :UmlNamespace   → UmlModelElement-set
26 constraint ∀nse UmlNamespace:
     |ns. ModelElement_namespace| = n, ne NAT, n ≤ 1
27 constraint ∀nse UmlNamespace-set, ∀ mse UmlModelElement-set:
       (∀ne ns: n.Namespace_owned =ms) ⇔ (∀me ms:
       m.ModelElement_namespace =ns)
28     Classifier_feature:UmlClassifier      → UmlFeature-sequence
29 Feature_owner:UmlFeature     → UmlClassifier
30 constraint ∀c eUmlClassifier, ∀ fse UmlFeature-sequence:
   c.Classifier_feature =fs ⇔ (f.Feature_owner = c, ∀fefs)
31 StructuralFeature_type:UmlStructuralFeature → UmlClassifier
32 BehavioralFeature_parameter:UmlBehavioralFeature → UmlParameter-
   sequence
33 Parameter_behavioralFeature:UmlParameter → UmlBehavioralFeature
34 constraint ∀b eUmlBehavioralFeature, ∀ pse UmlParameter-sequence:
              b.BehavioralFeature_parameter = ps
       ⇔      (p.Parameter_behavioralFeature = b, ∀peps)
35 Parameter_type:UmlParameter → UmlClassifier
36 Operation_method: UmlOperation → UmlMethod-set
37 Method_specification: UmlMethod → UmlOperation
38 constraint ∀ ms eUmlMethod-set, ∀ oe UmlOperation:
       o.Operation_method = ms ⇔ (∀me ms:
   m.Method_specification = o)
```

Fig. 3. ASM domains corresponding to UML meta-classes

The fact that a meta-class *inherits* another is expressed in ASM through an inclusion constraint between the domain corresponding to the parent class and the domain corresponding to the child class. Remark that here we are talking about the inheritance at *meta-model level*. While it would be of course extremely reductive to see inheritance at UML model level as simple domain inclusion, this is sufficient at

the meta-model level, where there are no operation definitions and the inheritance is added to make obvious semantic relationship between classes and to factorize specification. The ASM constraints (domain inclusions) corresponding to the generalization relationships from Fig. 2. are shown at lines 10-17 in Fig.3.

To each attribute and association navigable direction in the UML meta-model we associate an ASM function. For attributes, the function is defined on the ASM domain corresponding to the meta-class that owns the attribute, and with the codomain the ASM domain corresponding to the type of the attribute. Examples of functions corresponding to attributes are given at lines 18-23 Fig. 3. For associations, the functions are obtained similarly, as in the case of the functions at lines 24-36.

If the target association end has the multiplicity 1, the codomain of the function corresponding to that navigable direction is the ASM domain corresponding to the class connected to the target association end, as it is for instance the case with the functions at lines 29, 31, 35 in Fig. 3.

If the association end does not have the multiplicity 1 and if the target association end is ordered, then the codomain of the ASM function corresponding to that association direction is a *sequence* (ordered set) of the ASM domains of the target class, as in the functions at lines 28, and 32 in Fig. 3. Otherwise, the codomain is a new domain: a finite (unordered) set of the ASM domains corresponding to the target association end (ex functions at lines 24, 25, 27, 36 in Fig. 3.

In the case of bi-directional associations we add integrity constraints to capture the fact that the functions corresponding to each navigable directions are symmetrical. (lines 27, 30, 34, 38 in Fig. 3.)

Our formalization does not cover the entire UML. This means that not all the ASM names and functions obtained from the UML meta-model will be used subsequently. The ASM names and constraints not needed may be removed from the semantics or may be simply ignored. We choose to keep them in the semantics; this ensures that the information contained in ASM matches accurately the information contained in the UML meta-model, and makes further enhancements easier. Annex B of [17] contains most of the ASM code automatically obtained from the UML meta-model.

Additional Constraints

Besides the ASM constraints obtained after the translation of the UML meta-model, the static semantics of UML contains constraints corresponding to the well-formedness rules (WFR) existing in *UML Semantics* [19]. In the definition of UML, these constraints are expressed in OCL and we translate them, by hand, into ASM. We have managed to translate all the constraints written in OCL and some of the WFR that could not have been expressed in OCL (e.g. 2nd WFR of the SynchState).

Additional Functions

Next to the functions obtained from the UML meta-model, we define a set of functions that facilitate the description of the dynamic semantics: to access all the parents of a GeneralizableElement, to get all the attributes owned by a Classifier and having as *ownerScope* the classifier, to check if two operations have the same signature, etc.

Method Lookup

UML distinguishes between operations - signatures of the offered services - and methods –implementations for the operations. Several methods may correspond to the same operation. However, given a class and an operation defined in that class, there is only one *main* method that actually describes the body of the operation, the other methods being overwritten, as an operation may be defined in the ancestors of a class and may have a local method overriding its body.

We use a dynamic method lookup mechanism. Given an operation and a class, the method lookup mechanism gives the actual method that describes the body of the operation in the context of the considered class.

We use the method lookup mechanism existing in C++. This means that operations are identified by their signature (same as defined by UML, different of the method lookup used in e.g. Eiffel or SDL), and an operation is overwritten in an inheritor if it contains a local method with same signature. The method that implements a given operation is either a local method, or the method implementing the operation defined in the closest ancestor, that we find by traversing the inheritance relations captured from the meta-model. The whole lookup mechanism is embedded in a single function: *GetMethodForClassOperation*, this minimizes the impact of modifying the lookup mechanism. We disallow statically the conflicts generated by repeated inheritance.

```
GetMethodForClassOperation: UMLCLASS × UMLOPERATION → UMLMETHOD
GetMethodForClassOperation (c: UMLCLASS , op: UMLOPERATION ) = def
if ∃o∈c.Classifier_feature ∩ UMLOPERATION :HasTheSameSignature(o,op)
then    let o = take{o ∈UMLOPERATION|
                o∈ c.Classifier_feature ∧ HasTheSameSignature(o, op)}
        in o.Operation_method.main endlet
elseif ∃ directP ∈ UMLCLASS :(p ∈ c.GeneralizableElement_parent ∧
        ∃o1 ∈ p1.Classifier_feature ∩ UMLOPERATION :
HasTheSameSignature(o1, op))
        then    let <directP, op1>=take{<p,o> ∈ UMLCLASS × UMLOPERATION|
        p ∈ c.GeneralizableElement_parent
                ∧       o1 ∈ p1.Classifier_feature
                ∧       HasTheSameSignature(o1, op)}
                in o1.Operation_method.main                    endlet
elseif ∃ p ∈ UMLCLASS :(p ∈ c.GeneralizableElement_allParents ∧
                GetMethodClassForOperation(p, op) ≠ undefined)
        then let <p, m> = take{<p,m> ∈ UMLCLASS × UMLMETHOD|
                p ∈ c.GeneralizableElement_allParents
        ∧       m = GetMethodClassForOperation(p, op)
        ∧       m ≠ undefined }in m    endlet
else  undefined
endif
```

Fig. 4. Function describing the method lookup

3.2 UML Dynamic Semantics Structure

In this section we present the entities that concern run-time information and behavior. We complete the definition of the ASM names, started in the static semantics part, and we discuss how the ASM state changes at run-time. The dynamic structure is an

important component of the UML dynamic semantics, it contains the ASM domains and functions related to run-time.

The dynamic semantics defines the set of ASM agents and gives their corresponding transition rules. At this point, we focus on the behavior described by actions. We do not treat UML state machines, *i.e.* the dynamic semantics does not contain rules and agents corresponding to state machine, but we clearly identify the place of the state machines in the framework of the semantics definition. We formalize the mechanisms for signal passing, operation call, object creation and deletion, and the ASM program rules that corresponds to UML action executions.

Communication Mechanisms

Run-time UML objects interact by exchanging signals and operation calls. In UML the communication is point-to-point, this means that an object can send a signal to another object if it knows its identity. The communication semantics that we use for our formal definition, is based on the following principles: the communication is done by direct addressing, the signal and operation call passing may take time, the order of events is not necessarily preserved during the transport, though events are not lost. Active objects have incoming event queues, whereas passive objects do not.

The ASM communication modeling that we use is inspired by the communication modeling used in the SDL formal semantics [[13], [6]]. The communication is based on events passing. We add new properties to UML events: the *sender* (which is *undefined* for spontaneous events), the target, and the arrival time (a shared function that captures also the communication time).

In order to model the communication, we attach to each active object a gate (a concept needed for the formalization, not corresponding to anything in UML), which represents a unidirectional active object entry point, that exists at run-time. After an event is produced, it is placed in its target's gate, but it is made available to the target (it is part of the object's queue) only when the system time overpasses the event's arrival time.

This definition of queues and schedules does not constrain in anyway the queue policy. Any criterion could be applied for ordering the events in the *schedule* and in the *queue*, if needed ,some additional signal properties (ex. priority) could be defined.

Concurrency Model

In this section, we describe the basic principles of concurrency, as defined in our semantics. The concurrency handles aspects such as the relationship between threads and objects, characteristics of active and passive objects, relationship between operation call and state machine execution (even if we do not detail the state machine execution itself), etc.

The concurrency model that we describe subsequently is inspired by the UML concurrency model we have described in [14]. However, here we focus only on standard UML concepts, we do not add any non-standard concurrency concepts and we do not focus on state machine execution.

In our concurrency model, only active objects have thread(s) of control, while passive objects execute exclusively on the threads active objects. The consequences of these: passive objects do not have input events queue, asynchronous communication with passive objects is not possible, passive objects may not have

state machines, the only behavior mechanism existing in passive objects is contained in its operations (they can at most have protocol state machines, that only constrain their execution and do not describe it), when an active object calls an operation of a passive object, this operation executes on the callers thread.

In ASM, we model each execution thread using an ASM agent.

Active Objects

Each active object has an event gate and a main execution thread, modeled as ASM functions. The main thread manages the incoming events, ensures the dispatch of the events in the input queue and, depending on the nature of the dispatched events, relay them to the state machine or starts new operation executions, on stand-alone threads. If the active object has a state machine, then at least one of the threads owned by the object corresponds to the state machine execution[1].

A call directed towards an active object results in the creation of a new execution thread. The moment of its creation depends on the *concurrency* operation property (defined in [19] §2.5.), which specifies what happens if concurrent calls of the same operation exist. This attribute is mainly designed to protect passive objects against concurrent calls, but UML allows operations of active objects to use this property for specifying the semantics of concurrent calls. We consider that this attribute is useful for active objects, and we take it into account for the semantics of operation calls.

For exemplification, we take the trickiest case, which is when the call is directed towards a *guarded* operation (multiple calls are allowed, but the *call target* shall make sure that only one call is executed at a time, the others being blocked). An active object receiving a call to a *guarded* operation, first checks whether there are any other operations executions for the same operation running. If so, the operation call passes to a waiting status, until it can be executed. When no other operation execution exists for the called operation, a new thread is created on which the operation is executed. The ASM rule that describes these is given in Fig. 5. Not all the names used in this rule have been defined in this paper, nevertheless we tried to choose them as suggestive as possible. For their precise definition that reader is referred to [17].

Some basic rules that govern the behavior of active objects:

If the input queue of an active object is not empty, the active object successively treats all events contained in it, resulting in operation initiation, result return of remote calls towards the current object, or event forward to the state machine.

Since each call of an active object operation is treated on a different thread, recursive operation calls of are treated as regular calls, i.e. on several threads.[2]

The active object's state machine may receive call events, but only if they do not correspond to calls directed towards the current active object. This is to avoid the situations when the target object could ignore a call. When the main thread of the active object is available (no new events exist in the input queue), the active object may launch the calls towards guarded operations whose execution was blocked due to the *guarded* operation constraint, provided that they guard condition became valid.

---

[1] Concurrent sub-states of the state machine could be implemented on different threads, however this is a state machine semantics issue.

[2] One of the static constraints that we added, forbids recursive calls for *sequential* operations. Note however that this constraint cannot always be checked statically, thus our model cannot guarantee the absence of deadlocks, which we consider modeling errors.

```
TreatGuardedOperation(a:UmlActiveObject, op:UmlOperation, ev:UmlCallEvent)≡
if |Object_operationSlot(a, op).OperationSlot_execution|=0
    then
    if |Object_operationSlot(a, op).OperationSlot_waitingCalls|=0
    then
      LounchOperationThread(ev)
    Else
    let  callToLaunch = Object_operationSlot(a, op).
         OperationSlot_waitingCalls.head in
         LounchOperationThread(callToLaunch)
         Object_operationSlot(a, op).OperationSlot_waitingCalls :=
         (Object_operationSlot(a, op). OperationSlot_waitingCalls\
                  <callToLaunch>)⌢ <ev>       endlet
    endif
else
     Object_operationSlot(a, op).OperationSlot_waitingCalls :=
     Object_operationSlot(a, op).OperationSlot_waitingCalls ⌢ <ev>
Endif
```

Fig. 5. ASM Rule describing a call towards a guarded operation in an active object

Passive objects

Passive objects do not own execution threads; they may not have state machines and they do not have an input queue. There is however an implicit state associated to each passive object, this state given by the value of each attribute. The operations of a passive object execute on (one of) their caller's thread. A cascade of calls of passive class operations are all executed on the caller thread that initiated this call chain..

ASM Agents

During the execution we only have ASM agents corresponding to active objects. Each active object has a main thread that manages the event receipt, several threads corresponding to the ongoing operation execution and a (set of) thread(s) corresponding to the active object state machine, that we don't detail here). The agent corresponding to an operation executes the rule that corresponds to the main method of the operation. If this method is described using actions, then the ASM program rule of the agent that executes the operation is obtained from the action specification. If the operation's method is described using other means (state machine or other), then the specification of the behavior is outside the scope of our semantics definition, and we will consider that the agent executes a void program rule.

For each particular kind of action we define an ASM macro that gives its run-time semantics. In addition, we have a special rule that describes the action execution initialization and is executed right after an action starts to execute. It initializes all the functions related to the specific action.

Capturing Model Specific Information Through Initialization

Until now, we have presented the ASM elements that describe the static an dynamic semantics of UML. These elements are shared by all ASMs corresponding to UML

models[3]. We discuss here on the place of the UML model specific part in the semantics framework (see Fig. 1.).

The model specific part consists in specifying the content of each ASM domain at ASM start-time, using model specific information (e.g. populate the domain of classes, associations, attributes, action specifications, etc). The domains that correspond to run-time entities will be all empty at initialization time.

The initialization of names corresponding to dynamic behavior is done using *initially* integrity constraints. However, the system behavior initialization is a dynamic process and integrity constraints are not enough to capture its characteristics.

UML offers no standard mechanism for describing the system initialization. Various techniques could be imagined for capturing this information in UML (object diagrams, particular collaboration diagrams, deployment diagrams, etc.). However, as none of them is generally accepted, we are aware that the specification of the system initialization remains arguable. We assume that the UML model has a special object diagram, we call it *Initial* that gives the snapshot of the system at start time. This object diagram contains the objects and links that shall exist initially and the first operation calls.

Each ASM corresponding to an UML model, starts with an initialization phase. During this phase, a single ASM agent exists, we call it *Creator*, and this agent performs the following actions (in this order):

- it creates the objects described in the *Initial* object diagram. We do not add these objects directly to the ASM because the object creation may be followed by constructor operation call. Due to their definition, the agents corresponding to active objects are automatically created;
- it creates the links that were not created as a result of the object creation;
- it synchronizes the values of ASM names with the values of the fields in the object diagram;
- it creates call events corresponding to each call specified in the object diagram. These calls events will have the sender field unspecified;
- it dies.

When the *Creator* finishes execution, the system is in the status specified by the object diagram, and the call events corresponding to the initial calls are initiated. After the death of the creator the ASM behavior follows the rules described in the previous sections. The ASMs corresponding to UML models are assumed to live forever. However, as soon as no more active objects exist (*i.e.* no running agent), the execution of the ASM may be considered completed.

4 Related Efforts

A lot of effort was devoted to the goal of having a precisely defined semantics for UML. First, all UML tools that offer more than editing capabilities embed precisely

[3] Some optimizations may be imagined to eliminate the ASM names and rules irrelevant in the concrete case of the ASM corresponding to a precise UML model

defined semantics, *without making it explicit*. Code generators give semantics to classes, attributes, operations, etc, in terms of the target language. The semantics embedded in the code generators is incomplete, as in general only a part of the UML specification is translated to code. Symbolic executors (simulators) generally have a more complete semantics, which is embedded in the implementation of the simulator or it is expressed in the simulator language. In the case of UML tools, the semantics is not explicitly defined, and, in general, it is hidden in the tool implementation.

The academic tools using UML specifications, provide sounder semantics foundation, in general by focusing on particular aspects, e.g. USE [21] focuses on OCL constraints/ invariants, UMLAUT [12] focuses on formal validation based on protocol validation.

Other efforts to define precise semantics, also issued from the pragmatic need to fill the gap between advanced modeling concepts (such as state machines) and real implementation, are represented by the attempt to transform state machines into executable specifications. Notable results are described in [11].

The first attempts of formalizing UML date back in 1997 [1], when authors argued the need for a UML formal foundation based on a mathematical system model. This work is followed by more elaborated formalization efforts. The most notable of them are those done by the pUML group [[7], [8]], oriented towards formalizing UML in Z.

[9] formalizes UML state machines using graph transformation systems.

Other formalization attempts use ad-hoc semantics background, such as the collaboration semantics presented in [20], and of the action semantics base semantics [18].

Some UML formalization efforts using ASM already exist: [3] formalizes UML activity diagrams, by adapting ASM to natively support activity diagrams; [2] formalizes UML state machines by extending the basic ASM, with some new constructs to cover UML state machines specific features; [4] defines an ASM based toolset, focused on the state diagrams. These approaches disconnect state machines / activity diagrams of the rest of the language, and they do not consider all UML concepts (associations, inheritance, etc.).

[16] presents preliminary efforts of our study with focus on the static part.

5 Conclusions and Future Directions

We have used ASM to formalize the semantics of parts of UML that offer relevant information for the behavior description. Namely, we have described the semantics of classifier, class, data, expression, association, inheritance, operation, attribute, action description, and their run-time counterparts. The only operational behavior mechanism that we do not formalize is the state machine.

Our formalization approach consists in giving a precise method to obtain the ASM that corresponds to each UML model.

All the ASMs corresponding to UML models have a common part composed of: names (domains and functions) and constraints obtained from the UML meta-model, constraints corresponding to the WFRs from the UML specification and some additional integrity constraints; names (domains and functions), constraints and transition rules from the object model, action and operation run-time behavior,

definitions of the set of agents, and functions that give their behavior, from the object model.

This is the part that actually gives the semantics of UML. Each UML model can be fed into this semantics framework, to obtain the complete semantics description of each model. From a UML model we initialize the ASM by adding: initially constraints for the content of ASM names corresponding to meta-model entities obtained from the UML model. From the initial list of objects existing at initializations, obtained from the model's object diagram that gives the initialization status of the object. The same object diagram servers as basis for the definition of a *creator* agent that completes the initializations and creates the initial call events that start the system execution.

The research that resulted in the definition of UML semantics has various benefits: it fulfilled our initial goal, which was to define a precise semantics for actions, it proves that the actions defined in the previous section can be integrated not only into the UML meta-model, but more generally in the UML behavior framework, it unambiguously defines the semantics of UML constructs, it better underlines the relationships between various UML entities, it uncovers some UML inconsistencies and missing; it provides a complete UML object model, it uncovers issues to be treated when building UML-based tools, even outside the scope of this semantics, it proves that the goal of having precise definitions of UML ending into symbolic executions is realistic.

We see several directions on which this work could be continued in the future. First, we plan to integrate our semantics with an ASM tool. For the moment we have only managed to feed parts of our semantics and simulate very simple models. We intend to go forward on this direction. Another natural extension consists in adding the state machine semantics.

Another natural continuation of this work consists in taking into account more UML diagrams: check the run-time compliance of the behavior with UML sequence charts.

Currently a major revision of UML is ongoing, several proposals being submitted [22]. We plan to adapt our semantics to the new standard, as soon as this will be made available. For the static part this will be done almost automatically. For the dynamic part, we will have to review our semantics and check for its consistency with the new standard.

References

[1] R. Breu, et al.: Towards a formalization of the unified modeling language. In S. Matsuoka, M. Aksit (eds), ECOOP'97, In ECOOP'97 Proceedings, LNCS 1241, (1997)

[2] E. Börger, A. Cavarra, E. Riccobene. Modeling the Dynamics of UML State Machines, Int. Workshop on Abstract State Machines ASM'2000, LNCS 1912, (2000) 223-241

[3] E. Börger, A. Cavarra, E. Riccobene. An ASM Semantics for UML Activity Diagrams. In: T.Rust (Ed.), Proc. AMAST 2000, LNCS, Vol.1816, (2000) pp.292-308

[4] K. Compton, Y. Gurevich, J. Huggins, W. Shen.: An Automatic Verification Tool for UML, Technique Report CSE-TR-423-00, Dept. of EECS, The University of Michigan, May, (2000)
[5] S. Cook. The UML Family: Profiles, Prefaces and Packages. Proceedings of UML 2000 – The Unified Modeling Language. Advancing the Standard, LNCS 1939, (2000) 255-264
[6] R. Eschbach, U. Glässer, R. Gotzhein, M. von Lowis, A. Prinz: Formal definition of SDL-2000 – Compiling and Ronning SDL specifications as ASM Models. In E Börger, U. Glässer (Eds), Journal of Universal Computer Science, vol. 7 no 11(2001), 1024-1049
[7] Evans, R. France, K. Lano, B. Rumpe.: The UML as a Formal Modeling Notation. Proceedings of UML'98 – Beyond the Notation, LNCS 1618, (1999) 336-348
[8] Evans, S. Kent.: Core Meta-modelling semantics of UML: the pUML approach, Proceedings of UML'99 – Beyond the Standard, LNCS 1793, (1999) 141-155
[9] M. Gogolla, F. Parisi-Presicce.: State diagrams in UML: A formal semantics using graph transformation. In M. Broy, D. Coleman, T. Maibaum, B. Rumpe (Eds.), Proceedings PSMT'98. Technische Universitaet Muenchen, TUM-I9803, (1998)
[10] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, Specification and Validation Methods, pages 9-36. Oxford University Press, (1995)
[11] D. Harel, E. Gery. Executable Object Modelling with Statecharts, IEEE Computer 30(7):(1997) 31-42
[12] W. M. Ho, J. M. Jézéquel, A. Le Guennec, F. Pennaneac'h : UMLAUT: an extendible UML transformation framework. In *Proc. ASE'99, Florida*, October 1999
[13] International Telecommunication Union – Telecommunication Standardization Sector, Recommendation Z.100 Annex F - Specification and description language Electronic Bookshop, Geneva, (2000)
[14] Ober, I. Stan. On the Concurrent Object Model of UML, EuroPar'99, Toulouse, LNCS 1685, (1999), 1377-1384
[15] Ober, B. Coulette, M. Gandriau. Action Language for UML. LMO 2000, Mont Saint Hilaire, Canada, Hermes Science Publications, (2000) pp. 277-291
[16] Ober. More meaningful UML Models. Proceedings of TOOLS - 37 Pacific 2000, IEEE Computer Society Press, (2000), 146-157
[17] Ober Harmonizing Design Languages with Object-Oriented Extensions and an Executable Semantics. PhD thesis at Institut National Polytechnique de Toulouse, ENSEEIHT, Available at http://www-verimag.imag.fr/~iober/Thesis/thesis.html, April (2001)
[18] Object Management Group. UML Action Semantics Final Submission, OMG Document ad/01-03-01, 24 March, (2001)
[19] Object Management Group Unified Modelling Language Specification (UML), v 1.4, September (2001)

[20] G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language, Proceedings of UML'99 – Beyond the Standard, LNCS 1793, (1999) 99-115
[21] M. Richters, M. Gogolla: Validating UML models and OCL constraints. Proceedings of UML 2000 - Advancing the Standard, LNCS 1939, (2000), 265-277
[22] U2 Parteners Consortium. Revised UML 2.0 proposal for public review and comment, v 2.0 beta R2, June 2002

Privacy, Abstract Encryption and Protocols: An ASM Model – Part I

Dean Rosenzweig, Davor Runje, and Neva Slani

University of Zagreb

Abstract. We introduce an executable formal model of abstract encryption using the specification language AsmL, based on Abstract State Machines of Gurevich, providing a simple executable models for cryptographic protocols. We show strong universality properties of our descriptions of patterns, protocol roles and environment behaviors—no ASM program can do better, given the same information.

1 Introduction

We provide an executable formal model of abstract encryption using the specification language Asml [GSCG01], based on the Abstract State Machines model of Gurevich. The features of the ASM methodology enable us to avoid some difficulties of the more usual term–model of abstract encryption.

This seems to be the first formal model of abstract encryption that captures the nondeterministic character of encryption, and that can distinguish a situation where a known encryption is matched by identity, from the more usual situation where it is matched by its contents, through decryption. It also seems to be the first such model to capture, in an abstract setting, the notions of one–way functions, indistinguishability and nonmalleability, which are central to computational, probabilistic polytime cryptography, as local properties, native to the underlying formal framework—that of ASMs.

For a hierarchy of pattern classes we show rather strong universal properties: our patterns can do as well as any ASM program, given the same initial information, in both the job of extracting information from messages input, and in the job of creating messages to output. The composition and the semantics of pattern classes is given abstractly here, by an abstract syntax and deduction rules; the reader can examine and download their AsmL code from [web].

Out of patterns protocol roles can be composed. They are automatically executable. We show that any environment behavior with a multiset of such roles can also be described by a configuration of patterns. This means that a formal proof of attack impossibility, showing that a *description* of such behavior with patterns is impossible, also proves that no ASM program can mount an attack.

Implicit in these results is a model–checker for attacks: our patterns, protocol roles and environment behavior descriptions are running objects which can print and read themselves. We expect to present such a model checker explicitly soon.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 372–390, 2003.
© Springer-Verlag Berlin Heidelberg 2003

A way towards employment of theorem–proving technology of simple first–order logic with equality for security protocol analysis, suggested by our work, is hinted at in the concluding remarks, introducing Part II of this paper.

We build on the entire body of the ASM literature, as well as on formal studies of cryptographic protocols. In particular, G. Bella and E. Riccobene have used ASMs to analyze cryptographic protocols before [BR97, BR98]. They have however, unlike our model, reflected the usual term–models of abstract encryption. Our logical analysis in Part II, which drove Part I in a way, has been directly influenced by [DMP01], from where we borrow the usage of predicate source. Unlike their logic of actions, we are able to get by with simple first–order logic with equality.

Due to space limitations, we cannot introduce the reader very gently either to AsmL, ASMs or cryptographic protocols; we have to assume of the reader some familiarity with both bodies of literature.

We are indebted to the volume editors for being extremely forthcoming, and to anonymous referees for valuable comments.

2 Abstract Encryption

The notion of privacy of object–oriented programming languages of the C++ family, given some provisos, provides enough machinery for a faithful model of abstract encryption (on the abstraction level of the programming language). The provisos which should suffice are

1. the language has no facilities for explicit manipulation of addresses and (mis)usage of known memory layout;
2. the equality of class–objects is intensional, i.e. two newly created class–objects are never equal, unless an explicit redefinition of equality says so;
3. the language has a precise mathematical semantics.

The specification language AsmL, developed recently at Microsoft Research [GSCG01], satisfies all the above requirements. To get the idea, consider the small class in figure AsmL 1.

AsmL 1 Hash

```
sealed public class Hash
  private subject as Message
  public accept(m as Message) as Boolean
    return subject=m
  public override Equals(obj as Object) as Boolean
    match obj
      h as Hash: return h.accept(subject)
      _: return false
```

Whatever type `Message` might be, `hash`(m) = `new Hash`(m) defines a one-way function! It is a 1–1 function in view of redefined equality, it is easy to compute, but, within the abstraction level of the language, impossible to invert in general: unless an agent (program, process, ...) already has an access to the subject, it cannot obtain it through its hash.

It is well known that, as soon as we have one way functions, we also have strong cryptography [DK01].

The basic classes defined in figure AsmL 2 should suffice to convey the idea. To conserve some space, we do not treat hashes, timestamps and signatures in this paper, but they can (and will) be added in a smooth way. Intuitively, if

AsmL 2 Basic Classes

```
type Message= Key or Nonce or Encryption or Byte or Boolean or MessageSeq
sealed public class Nonce

sealed public class Encryption
  private k as Key
  private m as Message
  beDecrypted(d as Key) as Message?
    if k.accept(d) then return m else return null

abstract public class Key
  abstract function accept(k as Key) as Boolean
  function encrypt(m as Message) as Encryption
    return new Encryption(me, m)
  function decrypt(e as Encryption) as Message?
    return e.beDecrypted(me)
```

a program has access to an object e of class `Encryption` without having access to a key which would get accepted by $e.k$, it has no way of obtaining $e.m$. This behavior is ensured if the concrete subclasses of `Key` are sealed without any added functionality, cf. figure AsmL 3. Even more, the program has no way of distinguishing e from another `Encryption` with the same property—as far as no appropriate key is accessible, they are equally amorphous, and interchangeable.

This is, to the best knowledge of the authors, also the first formal model of abstract encryption that captures the nondeterministic character of encryption, and that can distinguish a situation where a known encryption is matched by identity, from the more usual situation where it is matched by its contents, through decryption. While not often seen in protocols, matching of encryptions by identity is not entirely inconceivable, say in order to thwart a replay attack. If the deterministic character of some encryption algorithms needs to be regained, an appropriate class of encryptions can have its equality overruled.

Given a precise mathematical semantics for the language, the above remarks can be made completely precise.

AsmL 3 Key Classes

```
sealed public class PrivateKey extends Key
  override function accept(k as Key) as Boolean
    match k
      pk as PublicKey: return pk.accept(me)
      _:               return false
  public opposite() as Key
    return new PublicKey(me)

sealed public class PublicKey extends Key
  private priv as PrivateKey
  override function accept(k as Key) as Boolean
    return priv=k
  public override Equals(pub as Object) as Boolean
    match pub
      pk as PublicKey: return pk.accept(priv)
      _: return false

sealed public class SharedKey extends Key
  override function accept(k as Key) as Boolean
    return me=k
  public opposite() as Key
    return me
```

The mathematical semantics for AsmL, in particular the treatment of its object–oriented aspects, is as yet somewhat implicit, scattered around the body of ASM literature [BG01, GGV02, GSV01, GSCG01], although no one doubts that we shall very soon see it spelled out in full detail. Thus our discussion here necessarily remains somewhat sketchy, and will have to be redone. Yet we believe that the semantics is understood well enough to be certain about our claims.

The semantics is based on a notion of state as a first–order structure with a background of (hereditarily finite) collections: sequences, sets and maps [BG00]. It is also often useful to view a state as a mapping from *locations* [GSCG01] to *values*, elements of the underlying universe.

The semantics supports notions of *reachability* through a value, and of *accessibility* to a class–object, both relative to a state, as follows:

Reachability of a value through another value is the smallest reflexive and transitive relation which is

- closed under support of collections (in case of maps, both the keys and the values);
- closed under components of structures;
- closed under public members of class–objects.

A value is *accessible* to a class-object o if it is reachable through

- values accessible as global in the corresponding declaration scope;
- data members of o, public or private.

We skip the complications due to `friend` and other access control of AsmL, since we don't use it in our models, we assume members to be either `public` or `private`. Since the `decrypt` function of class `Key` is a proper function, i.e. its evaluation has no side effects and its value depends only on its explicit arguments, we may safely add a closure condition including its values to the notion of accessibility:

- if k and e are accessible to o in $\mathcal{S}$, then the value of $k.\texttt{decrypt}(e)$ is reachable through e.
- if k is a decryption key reachable through x, then so is the opposite key $\overline{k}$.

The last statement reflects the fact that public key can in most schemes be extracted from (the usual representation of) the private one. This is directly represented by the function `opposite()`; the correspondence is 1–1 in view of the redefined equality of `PublicKey` .

Values accessible to o in a state $\mathcal{S}$ suffice to interprete all terms (in general excluding procedure and function calls) occurring in o's program (though not in general to calculate the updates), and thus constitute the (partial) *local state* of o in S, $\mathcal{S}_o$. If a value is accessible to o in $\mathcal{S}$, we shall also say that it is accessible in $\mathcal{S}_o$.

The above notions are closely related to the notions of relevant or critical object and local state of [BGS99, BG01].

Sketchy as the above might be, we are convinced that the fully spelled out semantics of AsmL will support the following

Claim. Let k be a key, m a message, $\mathcal{S}_o$ the local state of a class–object o in state $\mathcal{S}$.

1. Let, in $\mathcal{S}$, e be an encryption such that $k.\texttt{decrypt}(e) = m$. If neither k nor m is accessible in $\mathcal{S}_o$, then they are also inaccessible in the extended partial state $(\mathcal{S}_o, e)$ (a state o's program started with input e would start in).
2. Let k be a decryption key (shared or public) and m a message. If either $\overline{k}$ or m is inaccessible in $\mathcal{S}_o$, and remains inaccessible through possible submachine states created by o's program fired at S, the program of o fired at $\mathcal{S}_o$ cannot create an encryption e such that $k.\texttt{decrypt}(e) = m$.
3. Let, in S, e_1, e_2 be two encryptions inaccessible in S_o, so that no key k such that $k.\texttt{decrypt}(e_i) \neq \texttt{null}$ is accessible in S_o. Then there is an isomorphism of extended local states $(\mathcal{S}_o, e_i)$, $i \in \{1, 2\}$ which maps e_1 to e_2, and is an identity on accessible objects of $\mathcal{S}_o$.

The above properties are the faithful abstract counterparts of the classical probabilistic–polytime properties of encryption: 1. corresponds to *secrecy*, 2. corresponds to *nonmalleability*, and 3. corresponds to *indistinguishability*, [BDPR98]. We shall in the sequel often call the last property *indistinguishability*, not necessarily restricted to encryptions.

This is, to the best of the authors' knowledge, the first formal model of abstract encryption which can reflect the notions of secrecy, nonmalleability and

indistinguishability, central to computational cryptography, as immediate local properties, native to the underlying formal machinery—ASMs.

This correspondence frees us from several difficulties that the usual term–models of abstract encryption have with 'messages seen but not understood', [AR02, Syv00], and paves the way for future work on a smooth relation between abstract and computational encryption models. We shall discuss it in more detail elsewhere.

For full code of our basic model of abstract encryption, the reader might care to look at [web].

3 Patterns

Example 1. The notorious example of the Needham–Schroeder protocol is usually informally presented as follows:

$$\begin{aligned} a &\xrightarrow{\{a, n_a\}_b} b \\ a &\xleftarrow{\{n_a, n_b\}_a} b \\ a &\xrightarrow{\{n_b\}_b} b \end{aligned}$$

where agents are identified with their public keys, and encryption of message m under key k is depicted by $\{m\}_k$.

Such informal presentations, as well as most formal studies of abstract encryption [DMP01, FHG98, FHG99, FA01, CDL+00, DLMS02] and others, are, one way or another, about message patterns. Listening to the subject, we take the patterns seriously. In the first approximation, patterns are something like messages but can also contain variables. They are usually, in formal studies, represented by terms of some vocabulary. Here they also play some roles usually played by terms. However, since messages are not just ground terms, formal representation of patterns, faithful to their informal use, can hardly be construed just as terms. In the above informal diagram, the patterns convey messages both created for output and analyzed from input, and these two roles are not entirely symmetric.

In our executable model we have constructed a hierarchy of intelligent pattern classes, appropriate for the encryption model, which can both match and create messages (and print and read themselves etc). In this section we provide a succinct formal representation of their construction and semantics, and prove the essential features of what they can do. The idea is that *patterns suffice*: whatever information an (ASM) agent can extract from a message, a pattern can do as well, given the same information; whatever message an (ASM) agent can create, a pattern can do as well, given the same information.

When we compare capabilities of patterns to those of ASM programs, we assume that the programs are *isolated*: they start, possibly with an input, and in finitely many steps they terminate, possibly yielding an output, without communicating with the external world in the meantime. This means also not calling

any functions and procedures (except for `decrypt` and creation of objects, but all forms of iteration are allowed to them). The rationale is that, in a cryptographic protocol situation, all communication with the external world should be explicitly included in the protocol specification; what we have in mind is the internal steps between two communication points.

The interested reader can examine and download the full AsmL code for pattern classes from [web].

3.1 Abstract Syntax

The composition of pattern–classes can be succinctly represented by the following abstract syntax:

$$\begin{aligned}
p &\rightarrow createPat|\ bytePat \mid boolPat \mid [p_1, \ldots, p_n] \mid letPat \\
createPat &\rightarrow keyPat \mid noncePat \mid encryptPat \\
keyPat &\rightarrow eKeyPat \mid dKeyPat \\
dKeyPat &\rightarrow prv \mid sv \\
eKeyPat &\rightarrow \texttt{op}(dKeyPat) \mid pbv \\
noncePat &\rightarrow nv \\
bytePat &\rightarrow \texttt{Byte} \mid bv \\
boolPat &\rightarrow \texttt{true} \mid \texttt{false} \mid tv \\
encryptPat &\rightarrow \texttt{EP}(ev, eKeyPat, p) \mid ev \\
letPat &\rightarrow \texttt{let}\ v = p_1\ \texttt{in}\ p_2
\end{aligned}$$

where the patterns with names ending in 'v' represent the variables of appropriate types. In the `let` pattern the variable v is assumed to be fresh, occurring only in p_2. In the encryption pattern, the variable ev is a (somewhat clumsy) technical device only needed in order to retain in an assignment every object created and received; it involves a global syntactical constraint: this variable must occur uniquely, as the first argument of an `EP` construct, in a pattern. Of course, the encryption variable ev can not occur in pattern p_1. Whenever it is of no importance, we drop the variable ev in an encryption pattern.

3.2 Assignments and Matching

An *assignment* is a mapping of some pattern variables to messages of appropriate types. The idea is that an assignment represents the 'knowledge' of a (local) state, the collection of creatable messages (nonces, keys and encryptions) accessible to it. We shall sometimes use the AsmL notation $v \in \sigma$, where σ is an assignment and v a variable, with the meaning that v is in the *domain* of σ, and the *ad hoc* notation $\sigma.(v \rightarrow m)$ for the extension of σ mapping also v to m (given $v \notin \sigma$). Let us fix some terminology.

We shall say that a creatable message is *reachable* through σ (and maybe some messages) if the message is reachable through σ (and the messages) to any (local) state S to which σ (and the messages) are accessible.

A creatable message is *known* to σ if it is in the range of σ (we shall also consider a public key $\overline{k}$ to be known whenever its private counterpart k is); it

is *learned* by σ through a message m if it is not reachable through σ, but is reachable through σ, m.

An assignment σ is *complete* if all creatable messages reachable through σ are also known to σ; an assignment is complete for a state S if all creatable messages accessible to S are known to σ.

Two messages m and m' are *indistinguishable* for a complete assignment σ if they are indistinguishable for any (local) state σ is complete for.

A term–alike aspect of patterns is that appropriate assignments map them to messages.

Definition 1. *Let τ be an assignment and p a pattern. By simultaneous induction we define what is means for τ to be* appropriate *for p, and if it is, what is the message p^τ:*

- *if p is a constant c, then τ is appropriate for it and $c^\tau = c$;*
- *if p is a variable v then τ is appropriate for it if v is in its domain and $v^\tau = \tau(v)$;*
- *if p is $\mathtt{op}(v)^\tau$ then τ is appropriate for it if it is appropriate for v, and $\mathtt{op}(v)^\tau = \overline{v^\tau}$;*
- *if p is concatenation $[p_1, \ldots, p_n]^\tau$ then τ is appropriate for p if it is appropriate for all p_i, and $[p_1, \ldots, p_n]^\tau = [p_1^\tau, \ldots, p_n^\tau]$*
- *if p is an encryption pattern $\mathtt{EP}(ev, kp, p)$ then τ is appropriate for it if both*
  - *τ is appropriate for ev, kp and p_1, and*
  - *for some key k we have $kp^\tau = \overline{k} \wedge k.\mathtt{decrypt}(ev^\tau) = p_1^\tau$;*

  then $\mathtt{EP}(ev, kp, p)^\tau = ev^\tau$;
- *if p is $\mathtt{let}\ v = p_1\ \mathtt{in}\ p_2$, τ is appropriate for it if both*
  - *τ is appropriate for p_1, and*
  - *$\tau.(v \to p_1^\tau)$ is appropriate for p_2;*

  then $(\mathtt{let}\ v = p_1\ \mathtt{in}\ p_2)^\tau = p_2^{\tau.(v \to p_1^\tau)}$

Notice that it is an encryption variable in the encryption pattern which enforces the above consistency constraint.

If σ is an assignment, a pattern p is *consistent* with it if there is a $\tau \supseteq \sigma$ appropriate for p; a message m matches such a consistent pattern p over σ if, for appropriate $\tau \supseteq \sigma$, we have $p^\tau = m$. We shall call such τ a *matching witness* for σ, p, m.

There will be in general many patterns that a message m matches over σ. Is there a 'best' one in some clear sense? The answer depends on what we want to do with the message. Consider the following two scenarios:

1. An isolated program π in local state S with input m needs to extract some information learned through m. A pattern p would be 'the best' in this sense if no isolated program π could to better than p, given an assignment σ complete for S.
2. An isolated program π in local state S wants to output m. A pattern p would be 'the best' in this sense if it enforces creation of all objects that π starting from S needs to create.

We pursue both scenarios.

3.3 Patterns Suffice for Analysis

Let σ be an assignment, m a message. We say that m *opens boxes* in σ if σ knows some encryption e without knowing its decryption key, but learns such a key through m.

To factor out some complications, we first prove the theorem about analysis for the case of a message m that doesn't open boxes in σ.

Theorem 1. *Let σ be a complete assignment, m a message that doesn't open boxes in σ. Then there is a pattern p such that a message m' matches p over σ iff m' is, for σ, indistinguishable from m. Respective minimal matching witnesses τ, τ' are complete and also indistinguishable for σ.*

This means that no isolated program with input can analyze a message better than a pattern could. Since both σ and τ are complete here, knowing and learning obtain very sharp meanings: a message is known if it is in the range of σ; it is learned by the match if it is in the range of $\tau - \sigma$. One could also say that a submessage of m was *matched* if it is known.

Variables occurring in p can also be split into two categories: those *learned* from m by the matching, i.e. not in σ; and *matched* in m otherwise.

Proof. For σ, m we first construct the pattern p.

Pattern Construction. Associate to each creatable submessage c of m, a variable v_c of appropriate type, under the following provisos:

1. if c is known in σ, $c = \sigma(v)$, then v_c is v;
2. if c is a public key $\overline{k}$ not in the range of σ but is known via the known private key $k = \sigma(v)$, then v_k is v;
3. unknown public keys get a fresh *public* key variable;
4. other unknown creatable messages get a fresh variable;
5. repeated occurrences of unknown c get the same v_c.

Now construct a pattern p' by induction over m (doing some auxiliary jobs as well) as follows:

6. if m is a constant, p' is m;
7. if m is a public key $\overline{k}$ with an associated private key variable v_k, then p' is $op(v_k)$;
8. if m is other creatable message c, with associated variable v_c, then p is v_c.
9. if m is $[m_1, \ldots, m_n]$, then p' is $[p_1, \ldots, p_n]$, where p_i's are given by the induction hypothesis;
10. in case of unknown encryption learned from m, with the decryption key reachable through σ, m, an additional pattern p_e has to be created and associated to e: p_e is $\mathtt{EP}(v'_e, \mathtt{op}(v_k), p'_1)$, where v'_e is a fresh encryption variable and p'_1 exists by induction hypothesis. Notice that p'_1 does not contain any `EP` or `let` constructs. It is built from variables and possibly concatenations only.

This finishes the construction of p'.

11. Let $e_1, \ldots, e_n$ be the selected encryptions in m, in an ordering respecting the syntactical constraint on `let`, then

$$p = \texttt{let } v_{e_1} = p_{e_1} \texttt{ in } \ldots \texttt{let } v_{e_n} = p_{e_n} \texttt{in } p',$$
$$p_{e_i} = \texttt{EP}(v'_{e_i}, \texttt{op}(k_{e_i}), p'_i).$$

This finishes the construction of p.

Notice that all variables occurring in p, apart from those bound by `let`, are exactly the variables v_c associated to c which are either known to σ, or learned through m, or some v'_e for a selected encryption e.

Now let us construct $\tau - \sigma$. For all creatable submessages of m which are either known to σ or learned through m, and which are not the encryptions selected in proviso 10., set $\tau(v_c) = c$. For the selected encryptions, set $\tau(v'_e) = e$, where v'_e is given by proviso 10. By construction, τ is a complete extension of σ, and if it is a matching witness at all, then it is a minimal one.

To see that τ is a matching witness, see that all patterns p'_i and p' consist only of constants, variables either known to τ or shown in `let`, or concatenations. See further that each encryption variable v'_{e_i} occurs literally once in the pattern p, and is bound by τ to e_i. By induction we have: $\tau_i = \tau.(v_{e_1} \to e_1) \ldots (v_{e_{i-1}} \to e_{i-1})$ is appropriate for p_{e_i} for $i = 1, \ldots, n$ and $p_{e_i}^{\tau_i} = e_i$. Finally $\tau_{n+1} = \tau.(v_{e_1} \to e_1) \ldots (v_{e_n} \to e_n)$ is appropriate for p', $p'^{\tau_{n+1}} = m$ and $p^\tau = m$.

For some message m' indistinguishable from m, the same construction would work with the same pattern p and with the same variables, yielding τ' which is indistinguishable from τ for σ: each $\tau(v)$ is, by straightforward induction over its construction, indistinguishable from $\tau'(v)$ from σ.

The above described patterns will suffice for most 'normal' conditions. We have however to consider also some scenarios which are not inconceivable, though rarely, if ever, seen in protocols, such as obtaining an encrypted message, and, some time later, the decryption key.

Since we want to make strong completeness claims, we need to add an ability to express expectations on the form of encryptions learned but not yet understood (to be possibly understood in the future). To reduce such situations to the pattern–matching paradigm we need a notion of an extended pattern. An *extended pattern* is a pair $(p, \overrightarrow{v_e})$, where p is a pattern and $\overrightarrow{v_e}$ is a sequence of encryption variables.

An assignment τ is *appropriate* for an extended pattern $(p, \overrightarrow{v_e})$ if τ is appropriate for p and $p^\tau = [m, \tau(v_{e_1}), \ldots, \tau(v_{e_n})]$. Then we define the message $(p, \overrightarrow{v_e})^\tau$ as m.

Theorem 2. *Let σ be a complete assignment and m any message. Then there is a (possibly extended) pattern $(p, \overrightarrow{v_e})$ such that a message m' matches p over σ if and only if m' is for σ indistinguishable from m. Respective minimal matching witnesses τ, τ' are complete and also indistinguishable for σ.*

Proof. We only have to cover the case when m opens some boxes in σ; denote the opened boxes by $e_1, \ldots, e_n$, named in σ by $v_{e_1}, \ldots, v_{e_n}$. A message $m' = [m, e_1, \ldots, e_n]$ does not open any boxes in σ', which is σ with $v_{e_1}, \ldots, v_{e_n}$ removed. A message is reachable through σ, m if and only if it is reachable through σ', m'. By theorem 1, there is a pattern p and an complete assignment τ extending σ' such that $p^\tau = m'$. Notice that τ also extends σ since for each encryption e_i reachable through m' in σ', an appropriate variable v_{e_i} occurs in p. But then τ is appropriate for $(p, \overrightarrow{v_e})$ and $(p, \overrightarrow{v_e})^\tau = m$.

Corollary 1. *If σ can distinguish m from m', it can do so by a (possibly extended) pattern.*

3.4 Patterns Suffice for Synthesis

Let us turn to the scenario 2, that of creation.

We shall say that a creatable submessage c of message m gets *created* in m from σ if it doesn't occur in m only as a submessage of some message known to σ.

Theorem 3. *Let σ be a complete assignment and m any message. Then there is a pattern p matching m over σ such that the matching witnesses $\tau \supseteq \sigma$ is complete, and a minimal extension of σ which knows all messages created in m from σ.*

This means that no isolated ASM program with output can create a message that some pattern could not have created given the same knowledge. Since both σ and τ are complete here, matching and learning obtain duals, *forwarding* and *creating*: a submessage of m' if forwarded if it is in range of σ, and created if it is in the range of $\tau - \sigma$. Variables occurring in p can be classified in the same way. The essence of the crucial *source*–axiom of [DMP01] can in these terms be succinctly said: if you match a message you have created before, then someone has learned it.

Proof. We construct the pattern p by the same clauses as for analysis, in the proof of theorem 2, replacing provisos 3 and 10 by provisos 3' and 10':

3'. unknown public keys get fresh *private* key variables—we are imitating creation now, and it is impossible to create just a public key.
10'. like 10, but take *all* encryptions created in m from σ: if a key is missing, we shall create one;

If there are no encryptions created in m from σ, a straightforward induction proves the theorem. To see that it works in general, note that, by induction, when p_e^τ gets evaluated, it is with an assignment which already knows all encryptions properly contained in e.

3.5 Patterns Can Do by Themselves

In our executable model we have constructed a hierarchy of intelligent pattern classes, appropriate for the encryption model, which can both match and create messages (and print and read themselves etc). They have methods, `doCreate()` and `doMatch()` which execute theorems 1 and 2, that is compute the matching witnesses by themselves.

The interested reader can find the full code of patterns classes on the [web]. We represent these methods here abstractly, by deduction rules, and prove the relations of rules to theorems 1 and 3. The deduction rules can be seen as an abstraction the code, while the code can be seen as an implementation of the rules; in fact, they are a joint fixed point of an iterative process.

Matching Rules. An operational semantics of matching can be succinctly represented by the following deduction rules, with

$$\sigma, p, m \searrow \tau$$

meaning: with assignment σ the pattern p matches a message m, yielding new assignment τ. If no rule is applicable, the match fails. The output assignment extends the input one with the objects found on the way.

$$\sigma, c, c \searrow \sigma \tag{1}$$

$$\sigma, v, \sigma(v) \searrow \sigma \quad v \in \{sv, prv, pbv, nv, bv, tv, ev\} \tag{2}$$

$$\sigma, \mathtt{op}(dv), \overline{\sigma(dv)} \searrow \sigma \quad (dv : dKeyPat) \tag{3}$$

$$\frac{v \notin \sigma}{\sigma, v, c \searrow \sigma.(v \to c)} \tag{4}$$

$$\frac{\sigma, p_i, m_i \searrow \tau \quad \tau, [p_1, \ldots, p_{i-1}, p_{i+1}, \ldots, p_n], [m_1, \ldots, m_{i-1}, m_{i+1}, \ldots, m_n] \searrow \omega}{\sigma, [p_1, \ldots, p_n], [m_1, \ldots, m_n] \searrow \omega} \tag{5}$$

$$\frac{ev \notin \sigma \quad \sigma.(ev \to m), p, \sigma(dv).\mathtt{decrypt}(m) \searrow \tau}{\sigma, \mathtt{EP}(ev, \mathtt{op}(dv), p), m \searrow \tau} \tag{6}$$

$$\frac{ev \in \sigma \quad \sigma, p, \sigma(dv).\mathtt{decrypt}(\sigma(ev)) \searrow \sigma}{\sigma, \mathtt{EP}(ev, \mathtt{op}(dv), p), \sigma(ev) \searrow \sigma} \tag{7}$$

$$\frac{\sigma^-, p_2, m \searrow \tau \quad v \notin \tau}{\sigma, \mathtt{let}\ \ v = p_1\ \ \mathtt{in}\ \ p_2, m \searrow \tau.(v \to \sigma(v))} \tag{8}$$

$$\frac{\sigma^-, p_2, m \searrow \tau \quad \tau, p_1, \tau(v) \searrow \omega}{\sigma, \mathtt{let}\ \ v = p_1\ \ \mathtt{in}\ \ p_2, m \searrow \omega^-.(v \to \sigma(v))} \tag{9}$$

$$\frac{\sigma - \overrightarrow{v_e}, p, [m, \sigma(v_{e_1}), \ldots, \sigma(v_{e_n})] \searrow \tau}{\sigma, (p, \overrightarrow{v_e}), m \searrow \tau} \tag{10}$$

The assignments σ^-, ω^- above are respectively σ, ω with the variable v removed, to ensure the local scope of `let`. Matching rule (10) handles the extended pattern

case. We do not use, in our proofs, the power of the nondeterministic sequence rule, which allows backtracking in search for a successful match. We have included it nevertheless in order not to restrain the expressive power of the pattern language—without it, say if left–to–right matching were imposed, the pattern $[\mathtt{EP}(ev, \mathtt{op}(dv), p), dv]$ with unknown dv would not match any message, while the seemingly equivalent one $\mathtt{let}\ v = \mathtt{EP}(ev, \mathtt{op}(dv), p)\ \mathtt{in}\ [v, dv]$ could match; in our code, see [web], the nondeterminism is implemented by suspending an encryption without a known decryption key, and retrying it later.

Theorem 4. *Let σ be a complete assignment and m a message. Then there is (possibly extended) pattern p such that $\sigma, p, m \searrow \tau$, where τ is an analyzing assignment of theorem 1.*

Proof. In the view of theorem 2, we can limit the proof to the case of m not opening boxes in σ and a simple pattern p.

We construct a pattern p' in exactly the same way as in the theorem 1, except for one minor modification. Replace 11 by 11':

11'. like 11 but with an additional constraint on the ordering of encryptions e_i: if the matching k_{e_i} is unknown to σ, then it occurs in a pattern p' or p'_j for some $j > i$.

Notice that, because of syntactic constraint on `let` and by our construction, the following must also hold: if v_{e_i} does not occur in p' then v_{e_i} occurs in p'_j for some $j > i$. Such an ordering exists since all encryptions and decryption keys are reachable through σ, m.

We now prove the theorem by induction over the construction of m. The only interesting case is when m contains encryptions unknown to σ, with decryption keys either known to σ or learned by σ through m. In this case p is of form

$$\mathtt{let}\ v_{e_1} = p_{e_1}\,\mathtt{in}\ \ldots \mathtt{let}\ v_{e_n} = p_{e_n}\,\mathtt{in}\ p'.$$

To match p against m, according to `let` rule, we have first to match σ, p' against m, yielding τ_n; then match τ_i, p_{e_i} against e_i yielding $\tau_{i-1},\ i = n, \ldots, 1$.

See that, in the first step, matching p' against m, is the same as if all the e_i were black boxes: encryptions whose decryption keys are neither known to σ, nor learned by it through m. By induction over the uninteresting cases, this match succeeds, and τ_n is a complete assignment knowing about all creatable submessages of m *not* enclosed in encryptions there.

By the ordering of e_i's, both e_n and its decryption key is known to τ_n. Now iterate the induction hypothesis (since each e_i and its decryption key is in τ_i) yielding: τ_{i-1} is a complete assignment extending τ_i, knowing all the creatable submessages of e_i which are not enclosed in nested encryptions there; further, if $i > 1$, the decryption key for e_{i-1} is known to τ_{i-1}, $i = n, \ldots, 1$.

But then τ_0 is the τ of the theorem.

Creating Rules. An operational semantics of creating can be succinctly represented by the following deduction rules, with

$$\sigma, p \nearrow m, \tau$$

meaning: with assignment σ the pattern p creates a message m, yielding new assignment τ. Creation is not supposed to fail. The output assignment extends the input one with the objects created on the way. Of course, the output message is unique only up to inessential nondeterminism of `new`.

$$\sigma, c \nearrow c, \sigma \tag{11}$$

$$\sigma, v \nearrow \sigma(v), \sigma \tag{12}$$

$$\frac{cv \notin \sigma \quad c = \texttt{new T}}{\sigma, cv \nearrow c, \sigma.(cv \to c)} \quad c \in \{nv, sv, prv\}, \text{T appropriate} \tag{13}$$

$$\frac{\sigma, dv \nearrow \tau, k}{\sigma, \mathtt{op}(dv) \nearrow \overline{k}, \tau} \tag{14}$$

$$\frac{\sigma, p_1 \nearrow \tau, m_1 \quad \tau, [p_2, \ldots, p_n] \nearrow [m_2, \ldots, m_n], \omega}{\sigma, [p_1, \ldots, p_n] \nearrow [m_1, \ldots, m_n], \omega} \tag{15}$$

$$\frac{ev \notin \sigma \quad \sigma, kp \nearrow \tau, k \quad \tau, p \nearrow m, \omega \quad e = \texttt{new Encryption}(k, m_1))}{\sigma, \mathtt{EP}(ev, kp, p) \nearrow e, \omega.(ev \to e)} \tag{16}$$

$$\sigma, \mathtt{EP}(ev, kp, p) \nearrow \sigma(ev), \sigma \tag{17}$$

$$\frac{\sigma^-, p_1 \nearrow m_1, \tau \quad \tau.(v \to m_1), p_2 \nearrow m_2, \omega}{\sigma, \texttt{let } v = p_1 \texttt{ in } p_2 \nearrow m_2, \omega^-.(v \to \sigma(v))} \tag{18}$$

Theorem 5. *Let σ be a complete assignment and m a message. Then there is a pattern p such that $\sigma, p \nearrow m', \tau'$. For any complete assignment ω, to which no submessage, created from σ either in m or in m', is known, m is indistinguishable from m'.*

Proof. Take a pattern p as in the proof of theorem 3. The creation rules traverse m in a bottom-up manner, creating in fact, by invoking `new`, new versions of submessages created in m from σ. By rules and the construction of p every submessage needed is created only once and put in the right places. Since the respective submessages created in m and m' are fresh to ω satisfying the conditions of the theorem, there are also indistinguishable to it.

4 Modelling Protocols

Protocol roles are usually informally depicted as a sequence of input–output patterns, which is often directly reflected in the formal models. We follow the venerable practice, but in our terms. A nice side effect is that we can immediately execute the roles; the theorems of the preceding section delineate precisely the result of this execution.

We show that any environment interaction with a multiset of protocol roles can also be faithfully represented by a configuration of (extended) patterns.

By *any* we mean any, under some extremely general constraints, which outrule only true magic, jumping out of the abstraction level of the language. This means that impossibility of *configurations* expressing attacks of this or that kind, in fact implies impossibility of such *attacks* by any ASM programs. Since ASM programs are universal in several rather strong senses [Gur00, BG01], this universality property is as strong as one could hope for.

4.1 Protocol Roles

Definition 2. *A* role action *is a pair of patterns; a* role *is a sequence of role actions of fixed length* n, $[p_1 \Rightarrow q_1, \ldots, p_n \Rightarrow q_n]$. Current configuration *of a role is a pair of assignment and action index* (σ, i)*, denoted (for brevity) by* σ_i.

A role can be seen as an interactive machine with input and output of messages, with a program given by a sequence of actions. A configuration consists of state of memory (assignment) and program counter (action index). To specify a transition relation for such a machine, one specifies the relation of current configuration and input to next configuration and output, $\sigma_{i-1} \xrightarrow[out_i]{in_i} \sigma_i$:

$$\sigma_{i-1} \xrightarrow[out_i]{in_i} \sigma_i \equiv \exists \tau.\ \sigma_{i-1}, p_i, in_i \searrow \tau \ \wedge \ \tau, q_i \nearrow out_i, \sigma_i \tag{19}$$

Example 2. Roles of Needham-Schroeder protocol are as follows. Initially, variables pr_a and pr_b hold respective private encryption keys.

$$\begin{aligned} \text{Initiator} &= [b \Rightarrow \mathtt{EP}(b, [\mathtt{op}(pr_a), n_a]), \mathtt{EP}(\mathtt{op}(pr_a), [n_a, n_b]) \Rightarrow \mathtt{EP}(b, n_b)] \\ \text{Responder} &= [\mathtt{EP}(\mathtt{op}(pr_b), [a, n_a]) \Rightarrow \mathtt{EP}(a, [n_a, n_b]), \mathtt{EP}(\mathtt{op}(pr_b), n_b) \Rightarrow \mathtt{true}] \end{aligned}$$

Definition 3. *A* run *of a role in response to inputs* $in_1, \ldots, in_k$, $k \leq n$ *is defined as:*

$$\sigma_0 \xrightarrow[out_1]{in_1} \sigma_1 \ldots \xrightarrow[out_k]{in_k} \sigma_k.$$

The trace *of that run is* $\sigma_0, (in_1, out_1), \ldots, (in_k, out_k)$*. A run or trace is full if* $k = n$.

4.2 Environment Behaviors

A protocol is a set of roles, together with some wiring connecting output of one role with input of another. The idea of the wiring is to describe the desired message–transporting behavior of the environment.

Definition 4. *An* environment behavior *consists of a set of role–instances (multiset of roles)* $r_1, \ldots, r_k$*, together with their initial states* $\sigma_0^1, \ldots, \sigma_0^k$*, and a* behavior trace*: an initial state* σ_0*, and a sequence of role–tagged message pairs* $((out_i, in_i), r_j)$ *such that:*

(1) For every role r_j*, its initial state* σ_0^j*, together with subsequence of* r_j*–tagged message pairs* $((out_i, in_i), r_j)$*, is a trace of* r_j.

(2) Every message out_i can be created by an ASM program which has access to creatable messages only through σ_0 and messages $in_1, \ldots, in_{i-1}$ (except for the ones created by itself).

Since we view an environment as a black box, we should say that two environment behaviors are *indistinguishable* if for every role instance r_j involved the resulting role traces are indistinguishable to the initial state σ_0^j of r_j.

Security goals for protocols are often expressed as properties of sets of role configurations. For instance, one of Lowe's guarantees [Low97] can be expressed as follows: whenever the set contains a responder's role B in its final state, it also contains an initiator's role A in its final state, and they agree on some data in their assignments. Whatever a protocol goal might be, an *attack* can be defined as an environment behavior leading its roles to configurations forming a bad set, one not having the desired property. It is certainly reasonable to expect that protocol goal is invariant under isomorphism, in particular with respect to inessential nondeterminism of `new`.

An *environment behavior description* is just like an environment behavior, but with a trace description instead of a trace: replace messages (out_i, in_i) with patterns $(outPat_i, inPat_i)$, where $inPat_i$ can be in general also extended.

Environment behavior descriptions can be directly executed to compute environment behaviors. The algorithm maintains an assignment of pattern variables to messages, denoted by σ_i after step i. The step $i+1$ is:

- let r_j be the role with which ($outPat_{i+1}$, $inPat_{i+1}$) are tagged,
- compute $\sigma_i, outPat_{i+1} \nearrow out_{i+1}, \tau$,
- then write out_{i+1} to the input of r_j,
- then wait for in_{i+1} to appear on its output,
- then match $\tau, inPat_{i+1}, in_{i+1}$ to get σ_{i+1},

see [web] for code.

Example 3. The desired postman–like environment behavior for Needham-Schroeder protocol, and Lowe's attack on it, can be described with the following environment behavior descriptions, respectively. In both cases, take an instance A of the initiator with the initial state σ_0^A binding only a private key variable pr_a to its private key, and an instance B of the responder with initial state binding its private key variable pr_b. The initial state σ_0 binds variables a and b to public keys so that $\sigma_0(a) = \overline{\sigma_0^A(pr_a)}$ and $\sigma_0(b) = \overline{\sigma_0^B(pr_b)}$. The respective trace descriptions are then

$$\begin{aligned} \text{Postman} &= [((b, ev_1), A), ((ev_1, ev_2), B), ((ev_2, ev_3), A), ((ev_3, tv), B)] \\ \text{Lowe} &= [((\mathtt{op}(pr_e), \mathtt{EP}(\mathtt{op}(pr_e), [a, n_a])), A), ((\mathtt{EP}(b, [a, n_a]), ev_1), B), \\ &\quad ((ev_1, \mathtt{EP}(\mathtt{op}(pr_e), n_b)), A), ((\mathtt{EP}(b, n_b), tv), B)] \end{aligned}$$

where n_a, n_b are fresh nonce variables, ev_i are fresh encryption variables, tv is a fresh boolean variable, and pr_e is a fresh private key variable.

Since the notion of an attack is rather general, while the notion of attack description is quite specific, one might think that a restriction to environment behavior descriptions would limit the class of attacks considered. If for a given protocol security goal we prove that there is no attack description, what have we proved? In view of the universality properties of patterns, it is not surprising that we have proved that there is no attack whatsoever, at least not in the sense of the abstract encryption model.

Theorem 6. *For every environment behavior $\mathcal{B}$ there is a behavior description D with the same role instances and the same initial states such that any behavior $\mathcal{B}'$ computed by D is indistinguishable from $\mathcal{B}$ for the initial states.*

Proof. For each (out_i, in_i) and σ_i in $\mathcal{B}$ we proceed as follows: from σ_{i-1} and out_i by theorem 3 we create a pattern $outPat_i$ and a complete assignment τ_i; from τ_i and in_i we create an extended pattern $inPat_i$ and a complete assignment σ_i by theorem 2.

Let $S_{i,j}, S'_{i,j}$ be the local states or role r_j after the i-th action of the environment $\mathcal{B}, \mathcal{B}'$ respectively, represented by assignments $\sigma_{i,j}, \sigma'_{i,j}$. By induction over i, for all i and j, $\sigma_{i,j}$ is, for $\sigma_{0,j}$, indistinguishable from $\sigma'_{i,j}$.

Since patterns, protocol roles and environment behavior descriptions are implemented executable objects, a model checker for protocol behaviors is implicit in the above construction. We hope to present it explicitly very soon.

5 Concluding Remarks

Our work indicates that ASMs and AsmL can be very useful tools for modelling and investigating cryptographic protocols.

Since ASMs are implemented, ASM models of protocols are immediately executable; since patterns are universal, pattern–based ASM models are universal.

Known security proofs for simple protocols, such as Needham–Schroeder–Lowe, Perrig–Song, ... are easily reproduced by hand in our current framework. It would be more exciting if we could use the models for machine–assisted, if not outright automatic, proving of security properties for larger industrial–scale protocols.

This is precisely what we intend to do, and a part of the work will soon appear in Part II of this paper.

A small prerequisite is extending our treatment to hashes, timestamps and signatures, which is straightforward. A larger prerequisite for is plugging in logic, in a useful way. The predicates learned, matched, forwarded, created have the obvious interpretation, when their subjects are the individual actions, and the actions can automatically generate first–order formulae completely determining their (fixed–point) life. The individual actions are in fact the *proclets* of [BG01], and the whole environment behavior can be seen, in view of theorem 6, as a *ken* of [BG01]. It can automatically generate a first–order formula completely describing the behavior, up to indistinguishability from the initial states. This

formula is a close cousin of the Turing–tableaux description often used to prove the theorems of Church and Trakhtenbrot: if it is satisfiable at all, it is also satisfiable in an environment behavior. Together with a few simple facts about information flow, easily proved about any such ken, this sentence can be used to prove security properties with surprising simplicity, at least for the little examples we tried by hand—the key seems to be that the subjects of the basic predicates are the actions, and not, like usual, the agents, not even the roles. In Part II we explore the possibility of automatic generation of such formulae, and hooking them to theorem–proving technology.

6 Future Work

There are several further avenues to explore from here, which are not Part II but real future work. Two of them seem to us to be particularly exciting:

1. In [AR02] the bridge from abstract to computational cryptography has already been built, on the level of individual messages. It is not at all difficult to carry it over to our framework. It seems however that our framework is very convenient for an attempt to extend the bridge along the dynamics of protocol execution, represented here as a simple static structure. The goal this might achieve is a neat separation of concerns: to see that a protocol is computationally safe, in sense of probabilistic polytime, it should suffice to prove a) that it is abstractly safe, and b) that the cryptographic algorithms used are computationally safe.
2. We have analyzed matching of messages to patterns, in two ways: on input and on output. This might be extended to matching patterns to patterns, in a kind of directed unification, read–to–write and write–to–read. The results should be something similar to symbolic trace analysis of [FA01], but with an important addition: adding a new role into a behavior would also be a natural step, in a systematic search for any attack whatsoever. We know by [DLMS02] that this cannot possibly converge in general, since it would make security properties decidable; but finding sufficient conditions for convergence would extract decidable classes.

References

[AR02] Martin Abadi and Phillip Rogaway. Reconciling two views of cryptography. *Journal of Cryptology*, 15(2):103–127, 2002. 377, 389

[BDPR98] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public–key encryption schemes. In *CRYPTO '98*, volume 1462 of *LNCS*, 1998. 376

[BG00] Andreas Blass and Yuri Gurevich. Background, reserve, and Gandy machines. In *Proceedings of CSL'2000*, volume 1862 of *LNCS*, 2000. 375

[BG01] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. Technical Report MSR-TR-2001-117, Microsoft Research, 2001. 375, 376, 386, 388

[BGS99] A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*, 100(1-3), 1999. 376

[BR97] Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. *Journal of UCS*, 3(12):1337–1381, 1997. 373

[BR98] Giampaolo Bella and Elvinia Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*, 1998. 373

[CDL+00] Illiano Cervesato, Nancy Durgin, Patrick Lincoln, John Mitchell, and Andre Scedrov. Relating strands and multiset rewriting for security protocol analysis. In *Proceeding of 13th IEEE CSFW*, 2000. 377

[DK01] Hans Delfs and Helmut Knebl. *Introduction to Cryptography.* Springer, 2001. 374

[DLMS02] N. Durgin, P. Lincoln, J. Mitchell, and A. Scedrov. Multiset rewriting and the complexity of bounded security protocols. Technical report, 2002. 377, 389

[DMP01] Nancy Durgin, John Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *Proceeding of 14th IEEE CSFW*, 2001. 373, 377, 382

[FA01] M. P. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *Proceeding of 14th IEEE CSFW*, pages 160–173, 2001. 377, 389

[FHG98] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Honest ideals on strand spaces. In *Proceeding of 11th IEEE CSFW*, 1998. 377

[FHG99] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2/3), 1999. 377

[GGV02] Uwe Glaesser, Yuri Gurevich, and Margus Veanes. An abstract communication model. Technical Report MSR-TR-2002-55, 2002. 375

[GSCG01] Yuri Gurevich, Wolfram Schulte, Colin Campbell, and Wolfgang Grieskamp. *AsmL: The Abstract State Machine Language*, 2001. Version 1.5. 372, 373, 375

[GSV01] Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Toward industrial strength abstract state machines. Technical Report MSR-TR-2001-98, 2001. 375

[Gur00] Yuri Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000. 386

[Low97] Gavin Lowe. A hierarchy of authentication specifications. In *Proceedings of 10th IEEE CSFW*, 1997. 387

[Syv00] Paul Syverson. Towards a strand semantics for authentication logic. In *Electronic Notes in TCS*, volume 20, 2000. 377

[web] Web. `http://www.fsb.hr/~drosenzw/protocols`. 372, 377, 378, 383, 384, 387

A Framework for Modeling the Semantics of Expression Evaluation with Abstract State Machines

Wolf Zimmermann[1] and Axel Dold[2]

[1] Martin-Luther Universität Halle-Wittenberg
Institut für Informatik, 06099 Halle/Saale, Germany
zimmer@informatik.uni-halle.de
[2] Universität Ulm
Abteilung Künstliche Intelligenz, 89069 Ulm, Germany
dold@informatik.uni-ulm.de

Abstract. We present a framework for formalizing the semantics of expression evaluation using Abstract State Machines. Many programming languages allow some non-determinism for evaluating expressions. The semantics only have in common that arguments are evaluated before an operator is applied. The evaluation of one argument may be interleaved with the evaluation of the other arguments. However, programming languages usually restrict this most liberal evaluation order. For example, the expression evaluation may take into account short-circuit evaluation of boolean expressions which implies that right operands must not be evaluated before the complete left operand is evaluated. Our approach provides a generic expression evaluation semantics that only need to be instantiated adequatly. We demonstrate this approach by the example of Ada95, C, C++, Java, C#, and Fortran.

1 Introduction

Programming language definitions specify some non-determinism in the semantics of expression evaluation. However, real-life programming languages do not allow full non-determinism. Programming languages with short-circuit evaluation of boolean operators usually specify that the left operand is evaluated first. Then, depending on the result of the evaluation of the left operand, the right operand is evaluated. For example, if the left operand of a logical and-operator evaluates to false, the right operand must not be evaluated and the overall result of the expression is false. For any programming language, the evaluation order of expressions is specified such that it is compatible with the data-flow – even if the evaluation may cause side-effects.

For languages with assignment expressions (e.g. C and C++[12]), the evaluation order is also liberalized w.r.t. the execution of the updates specified by assignments. In particular, these updates can be executed in any order between some pre-specified program points. The main reason for a liberal expression evaluation order is efficiency since a compiler may choose any of the legal evaluation

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 391–406, 2003.
© Springer-Verlag Berlin Heidelberg 2003

orders. However our goal is not to assess whether this a good or bad practice to define expression evaluation as above but simply to model it. Ada [10] defines expression evaluation similarly. Other languages, e.g. FORTRAN [11], define that it is illegal if the the result of expressions depend on the order of their evaluation. This property is not decidable but should be expressed as a predicate over a formalization. This should be expressed as a constraint on runs. Furthermore, the results of expressions in statements executed later may be affected by the evaluation order of previous expressions. Hence, even for those languages, the non-determinism for expression evaluation must be formalized.

These restrictions and liberalizations are the reason that the typical formalizations of non-deterministic expression evaluation with structural operational semantics do not model the semantics of real-life programming languages such as C, C++, Ada etc. Furthermore, there are formalizations of programming language semantics using abstract state machines, e.g. C [7] and Java [16, 14]. However, the formalizations for expression evaluations [7, 14] look rather different and cannot be related to each other.

Some programming language definitions do not use the term "non-deterministic" for specifying the expression evaluation order. Instead, a typical specification states that this order is "implementation-dependent". In practice, this means that compilers or interpreters may choose any evaluation order. However, it is not the case that a compiler fixes a general rule of an evaluation order and then uses this order for each expression in the program. Instead, it may choose for every expression a different order. For an optimizing compiler the situation becomes even worse, because there are optimizations that copy code (e.g. code motion). Each copy of an expression may be evaluated using different evaluation orders. Hence, from a programmer's viewpoint the program behaves non-deterministically. Therefore, even if a language definition states that the order of expression evaluation is implementation dependent, it is modeled adequately with non-determinism, since programmers cannot rely on any particular evaluation order other than specified in the language definition.

The main goal of the paper is to provide a framework for ASM-formalizations for expression evaluation. We demonstrate the approach by formalizing the expression evaluation semantics of Ada [10], C [13], C++ [12], Java[3], C# [2], and Fortran[11]. To the best of our knowledge, this is the first framework of this kind of non-deterministic expression evaluation.

Section 2 briefly introduces abstract state machines, in particular those notations used in this paper. In Section 3 we describe the general notions for formalizing the semantics of imperative programming languages. Section 4 details the expression evaluation of C and demonstrates the freedom and restrictions for the evaluation order. The following Section 5 demonstrates why the classical approaches for non-deterministic expression evaluation from structural operational semantics do not work and gives an overview over the ASM-literature for programming language semantics. Section 6 defines the framework for expression evaluation. Finally, Section 7 shows the instantiation of this framework for

the languages Ada (Pascal and Modula are similar), C, C++, Java, C#, and Fortran.

2 Abstract State Machine

We first define the notion of Abstract State Machines. For more details, we refer to the Lipari-Guide [4] and the ASM 1997 Guide [5]. An *Abstract State Machine* (ASM) is a tuple $(\Sigma, \Phi_{\text{Init}}, \mathit{Trans})$ where Σ is a signature, Φ_{Init} is a set of Σ-formulas (the *initial conditions*), and *Trans* is a finite set of *transition rules*. The set of *states* is the set $\text{Alg}(\Sigma)$ of Σ-algebras; $[\![\cdot]\!]_q$ denotes the interpretation function of symbols of Σ in Σ-algebra q. A state q is *initial* iff q is a model of Φ_{Init} in the sense of logic, denoted as $q \models \Phi_{\text{Init}}$. In this article, we use order-sorted partial Σ-algebras.

Notation: Sorts are denoted by capital letters, function symbols always start with a lower-case letter. $S_1 < S_2$ denotes that S_1 is a subsort of S_2. The symbol $f : T_1 \times \ldots \times T_n \to T$ denotes a function symbol representing a total function and $f : T_1 \times \ldots \times T_n \to ?T$ denotes a function symbol representing a partial function. $t \in S$ is satisfied iff t is a term of sort S. $t[x/t']$ denotes the term t where each occurrence of variable x is substituted by term t'. This can be extended as usual to Σ-formulas (here, only free variables are substituted).

Transition rules are specified as in [5]. In this paper, we use the following kinds of transition rules:

if φ **then** *Transitions* **endif**
if φ **then** $\mathit{Transitions}_1$ **else** $\mathit{Transitions}_2$ **endif**
choose $\mathit{var} : \mathit{SORT} \bullet \varphi$
    Transitions
endchoose

where φ is formula of many-sorted first order predicate logic and *Transitions* is a set of transitions of one of the above forms or an update $f(t_1, \ldots, t_n) := t$. Intuitively, an update changes the interpretation of f: if the state transition from state q to state q' executes this update, then

$$[\![f]\!]_{q'}(a_1, \ldots, a_n) = \begin{cases} [\![t]\!]_q & \text{if } a_1 = [\![t_1]\!]_q, \ldots, a_n = [\![t_n]\!]_q \\ [\![f]\!]_q(a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

The first kind of transition rule executes the transition rules of *Transitions* in a state q iff $q \models \varphi$. Similarly, the second kind of transition rule executes the transition rules of $\mathit{Transitions}_1$ in a state q iff $q \models \varphi$. Otherwise, the transition rules of $\mathit{Transitions}_2$ are executed in a state q. The third kind of transition rules executes the transition rules $\mathit{Transitions}[\mathit{var}/t]$ in state q iff t is a term of sort *SORT* such that $q \models \varphi[\mathit{var}/t]$. If more than one such term exists, any of these terms is chosen non-deterministically. Macros can be used to abbreviate updates and terms. It is just a parameterized textual substitution mechanism. A macro definition has the form $\mathit{macro}(\mathit{par}_1, \ldots, \mathit{par}_k) \triangleq \mathit{text}$. If in an ASM-specification a term $\mathit{macro}(\mathit{arg}_1, \ldots, \mathit{arg}_k)$ is used, then it is textually replaced by $\mathit{text}[\mathit{par}_1/\mathit{arg}_1, \ldots, \mathit{par}_k/\mathit{arg}_k]$.

3 Programming Language Semantics

The semantics of programming languages is based on the abstract syntax rather than the concrete syntax. Abstract syntax is defined by a context-free grammar and defines an abstract data type, called abstract syntax trees. The nodes of such trees carry semantic meaning and the meaning of a program is composed from the meaning of these elements. The abstract syntax normalizes the concrete syntax in several respects: it no longer cares about representational details such as use of keywords or rules for operator precedence etc. These details are encoded either in node names of subtrees or in the structure of an expression tree. Also selection rules such as `statement ::= assignment | procedure call | loop | ...` which only enumerate the possible alternatives, are avoided: whenever a statement is allowed then one of the alternatives is directly inserted in the abstract syntax tree. Thus, the tree only contains leaves and so-called *composition productions* composing a language element from smaller units (subtrees). Every tree element has a semantic meaning which influences program execution. It is easy to represent abstract syntax trees as order-sorted algebras. Each non-terminal represents a sort. Each composition production $A ::= X_1 \cdots X_n$ represents a constructor $c : X_1 \times \cdots \times X_n \to A$, and each selection rule $A ::= X_1 \mid \cdots \mid X_n$ represents subsort relations $X_1 < A, \ldots, X_n < A$.

Static semantic is concerned with evaluating context conditions i. e. name analysis, type checking/inference, and operator identification, and checking their consistency. It works on symbolic information and does not care about the dynamic interpretation of a program. Whether an operation named *integer addition* really means an addition (not a subtraction) is of no concern. Traditionally semantic analysis is described by an attribute grammar, based on abstract syntax. An abstract syntax tree with computed attributes is called an *attributed syntax tree*. Any dynamic semantics is based on correctly attributed syntax trees. Hence, in terms of algebras, a program is a term of the sort of the start symbol Z. An attribute a of sort T may be different for different occurrences of a subterm t in the same program. Hence, occurrence algebras must be considered. As usual, an occurrence is a list of natural numbers. Let t be a term. If o is the empty occurrence, then the *sub-term of t at occurrence o* is t itself. If $f(t_0, \ldots, t_{n-1})$ is the *sub-term of t at occurrence o*, then t_i is the *sub-term of t at occurrence $o.i$* for $i = 0, \ldots, n-1$.
Notation: OCC denotes the sort of occurrences. $occ(t, o)$ denotes the sub-term of t at occurrence o. In an abstract syntax tree, we have an attribute *occ* denoting the occurrence of a node. The attribution rules computing this attribute are straightforward.

An attribute a of type T associated with non-terminal X can be modeled as a partial function $a : Z \times OCC \to ?T$. It is defined iff $occ(t, o) \in X$. Attribution rules become equations that are axioms and conditions also become axioms.
Notation: $x.a$ denotes the attribute a of the node specified by occurrence x.

Any imperative programming language defines instructions which are nodes of the abstract syntax tree. We call these instructions *tasks*. Control-flow and data-flow is defined by special attributes. The attribute $nt : \{OCC\}$ specifies

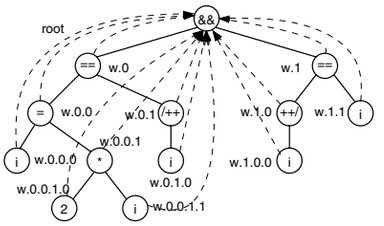


Fig. 1. Attribution of an Expression

Table 1. Attributes for Expression Evaluation

$occ : OCC$	occurrence of each AST-node
$nt : \{OCC\}$	set of next tasks that can be executed
$dest, src : OCC$	destination/source of an assignment
$lop, rop : OCC$	left/right operand of an operator
$root : OCC$	occurrence of the root of the expression
$parent : OCC$	occurrence of the parent node
$subExprsOut : \{OCC\}$	all occurrences of sub-expressions rooted at current node
$subExprsIn : \{OCC\}$	all occurrences of nodes belonging to *root*

the set of possible tasks to be executed next. Table 1 summarizes the attributes required for expression evaluation. Figure 1 shows an example expression $((i = 2 * i) == i++)\&\&(++i == i)$. The numbers show the occurrences of the nodes provided the root of the expression has occurrence w. The operators are denoted within the nodes and represent their types, respectively. The dotted arrows denote the root attribute. The other attributes shown in Table 1 are straightforward.

We define the state space of imperative languages. An abstract state machine semantics has an instruction pointer that points to the program π to be executed, a memory, and a runtime environment that stores for each active procedure call the bindings for local variables, the static predecessors, and the return address (i.e. the instruction calling the procedure), cf. Fig. 2.

Hence, we have the following dynamic functions:

$pc : OCC$	program counter
$mem : LOC \to VALUE$	memory, $VALUE$ includes all values of the language
$env : ENV$	runtime environment

Notation: The macro $ct \triangleq occ(\pi, pc)$ denotes the sub-tree of π given by the current task. Macro o **is** $TYPE \triangleq occ(\pi, o) \in TYPE$ denotes that the node pointed to by o is of a certain sort.

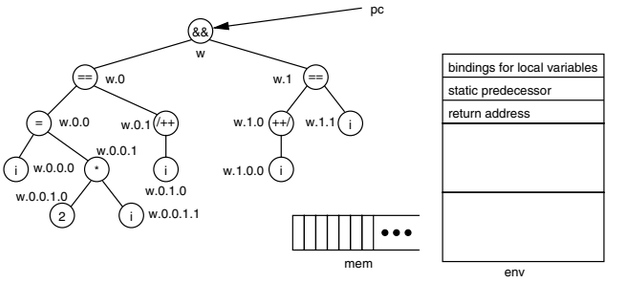


Fig. 2. State Space of Imperative Programming Languages

4 An Example: Expression Evaluation in C

The C language standard [13] defines evaluation orders as follows:

6.5.2.2 Function Calls (10): *The order of evaluation of the function designator, the actual arguments, and the subexpressions within the actual arguments is unspecified, but there is a sequence point before the actual call.*

Here, the non-deterministic evaluation order is explicitely mentioned. Furthermore, it states that even the subexpressions of the arguments can be evaluated in any order and that the function call is executed after the designator and all arguments are evaluated. Most binary operators in the C language standard [13] are defined similarly. The C-language definition allows even more freedom for expression evaluation, when assignment expressions are present:

6.5.16 Assignment Operators: Semantics *An assignment operator stores a value in the object designated by the left operand. An assignment expression has the value of the left operand after the assignment, but is not an lvalue. ... The side effect of updating the stored value of the left operand shall occur between the previous and the next sequence point. The order of evaluation of the operands is unspecified.*

If several assignments occur in a sub-expression, their updates can be executed in any order between two sequence points – independent of the data-flow. Sequence points are for example roots of expressions, the roots of the left operands of the binary logical operators && and ||, function calls (w.r.t. their arguments), the root of the condition in conditional expressions, and the comma operator.

C [13] defines short-circuit evaluation for boolean expressions. For example:

6.5.13 Logical AND Operator Semantics

3. *The* `&&` *operator shall yield 1 if both of its operands compare unequal to 0; otherwise, it yields 0.*
4. *Unlike the bitwise binary* `&` *operator, the* `&&` *operator guarantees left-to-right evaluation; there is a sequence point after the evaluation of the first operand. If the first operand compares equal to 0, the second operand is not evaluated.*

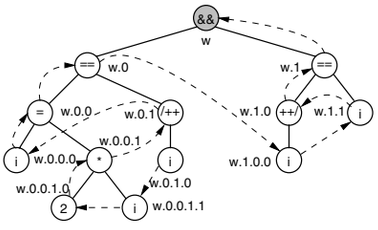


Fig. 3. An Expression of C

This evaluation order must be combined with the other definitions for expression evaluation order. In particular, this combination implies that any sub-expression of the left operand of a binary boolean operator is evaluated before the corresponding right operand is evaluated. However, if the right operand is evaluated, then its evaluation may be interleaved with the evaluation of other sub-expressions of the same expression. Conditional expressions behave similarly.

The following example shows some evaluation orders for expressions in C:

Example 1. Consider the expression in Figure 3. The C semantics only requires that the left sub-expression of && is evaluated before the right sub-expression, that the updates of the assignments at $w.0.0$ and $w.0.1$ (post increment) are executed before computing the value of the left operand, and that the assignment at $w.1.0$ (pre increment) is executed before the value of the expression is executed. The left-to-right evaluation order is a legal evaluation order according to C:

$w.0.0.0, w.0.0.1.0, w.0.0.1.1, w.0.0.1, w.0.0, w.0.1.0, w.0.1, w.0, w.1.0.0, w.1.0, w.1.1, w.1, w$

If the execution is started with $i = 2$, after execution, the result of the expression is 1 and $i = 6$.

The arrows show a rather different evaluation that is also legal:

$w.0.1.0, w.0.0.1.1, w.0.0.1.0, w.0.0.1, w.0.1, w.0.0.0, w.0.0, w.0, w.1.0.0, w.1.1, w.1.0, w.1, w$

When evaluating the left operand according to this order, its value becomes 0. Hence the right operand is not evaluated. If the execution starts with $i = 2$, the result of the expression is 0 and, after the execution, it is $i = 4$.

5 Related Work

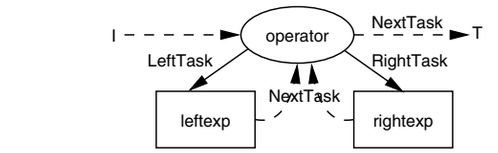
Non-deterministic expression evaluation can be easily expressed using structural operational semantics and expression contexts. Typically, it can be formalized using the inference rule

$$\frac{\langle e, \sigma\rangle \to \langle v, \sigma'\rangle}{\langle \mathcal{E}[e], \sigma\rangle \to \langle \mathcal{E}[v], \sigma'\rangle}$$

Here $\mathcal{E}$ is an expression with a "hole", the context of an expression e. The rule is interpreted as follows: if an expression e evaluates to a value v while changing the memory σ to σ', then any expression $\mathcal{E}[e]$ evaluates to $\mathcal{E}[v]$ while changing the memory σ to σ'. I.e., the context is unchanged and the sub-expression e is replaced by the value v. Hence, the evaluation of an expression can evaluate any sub-expression. Thus, this rule specifies fully non-deterministic evaluation order.

Unfortunately, structural operational semantics becomes rather complicated when the evaluation orders are changed w.r.t. full non-determinism. Norrish [9] tries to extend this rule for formalizing the semantics of C. His formalization of expression evaluation in C needs 36 inference rules and requires many new definitions and concepts. It is not obvious that the formalization correctly specifies the expression evaluation semantics of C.

Many ASM formalizations of programming languages and their implementations can be found in the literature. This is one of the main applications of




```
EVALUATE OPERANDS WITH(stats) ≜
    if Visited(CurTask) = neither then
        if ChooseTask(CurTask) = LeftTask(CurTask) then
            Visited(CurTask) := left
        elsif ChooseTask(CurTask) = RightTask(CurTask) then
            Visited(CurTask) := right
        endif
        Moveto(ChooseTask(CurTask))
    elsif Visited(CurTask) = both then
        Visited(CurTask) := neither
        stats
    endif
```

Fig. 4. Expression Evaluation of C defined in [6]

ASMs e.g. [6, 7, 1, 14, 15, 8]. The formalizations of the C language both overspecify the evaluation order of expressions [6, 7]. According to these formalizations, one operand is evaluated completely before another operand is evaluated. E.g. for binary operators either the left operand is evaluated first and then the right operand is evaluated or vice versa. Figure 4 shows the corresponding macro taken from [6]. *ChooseTask* is an external function that non-deterministically selects the left or right operand. According to the control-flow, if one operand is started to be evaluated, it is not left before its evaluation is complete. The specification using Montages by Huggins and Shen [7] similarly defines expression evaluation. The formalization of the C++-language [15] extends the C-semantics [6] to C++. Expression evaluation is defined as for C. The formalization of the semantics of Oberon [8] specifies an evaluation order from left to right.

6 A Generic Approach for Formalizing Expression Evaluation

The basic idea is

- first, to define a binary predicate $\prec$ on occurrences specifying which sub-expressions must be evaluated before other sub-expressions,
- second, to non-deterministically choose a topological order of the occurrences of an expression w.r.t. to the transitive closure $\prec^+$ of $\prec$ whenever this expression is started to be executed,
- third, to evaluate this expression according to the topological order, and
- fourth, if operators (such as e.g. the logical *and* in C) may suppress the evaluation of sub-expressions, the occurrences of these sub-expressions are removed from the topological order when it is decided that they are not evaluated.

This approach has several consequences:

1. The values of sub-expressions have to be stored. These values must be stored in the frames of the runtime environment because expressions might contain function calls. We model these by the dynamic function $val : ENV \times OCC \to VALUE$.
2. A list of occurrences of sub-expressions that have still to be evaluated must be maintained. Again, because of function calls within expressions, this list has to be stored in the frames of the runtime environment. Therefore, it is modeled by the dynamic function $toBeEval : ENV \to [OCC]$.
3. It must be specified when an expression is entered and how to advance the program counter, if an instruction is executed that does not belong to an expression and if an instruction is executed that belongs to an expression.
4. The evaluation order $\prec$ must be specified.
5. It must be specified which sub-expressions (at occurrence o') are evaluated if another sub-expression at occurrence o evaluates to a value v. This is formalized by a predicate $WillBeEvaluated(o, v, o')$. The macro $NotEvaluated(o, v) \triangleq$ $\{o' | o' \in o.parent.subExprsOut \wedge \neg WillBeEvaluated(o, v, o')\} \setminus \{o.parent\}$ specifies the occurrence of the sub-expressions not being evaluated if the sub-expression at occurrence o evaluates to v.
6. Finally, the updates of an operator must be specified.

In this section, we show that the first three issues can independently be formalized of the specific programming language. The formalization of the three other issues is language dependent. The first two consequences lead to a state space illustrated in Fig. 5. For further formalizations, it is convenient to define macros for the values in the current environment and the occurrences of the sub-expressions of the current environment:
$value(o) \triangleq val(env, o)$ and $toBeEvaluated \triangleq toBeEval(env)$
An expression will be entered for evaluation at occurrence t, if the task of t is an expression and, either the current task being executed is not an expression or

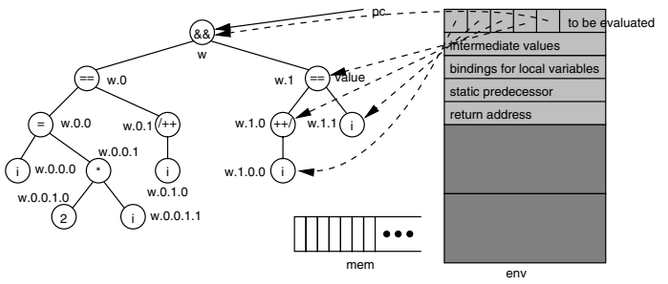


Fig. 5. Dynamic Functions taking into account Expression Evaluation

a different expression. The latter can be recognized by the roots of these two expressions, since these roots must be different. Hence, the following macro defines when an expression is entered for evaluation:

$EntersExpr(t) \triangleq (\neg(pc \textbf{ is } EXPR) \vee pc.root \neq t.root) \wedge (t \textbf{ is } EXPR)$

Advancing the program counter pc if it is not an expression simply chooses one of its next tasks. If this task t enters an expression e then a topological order l of the occurrence of the sub-expressions of e w.r.t. $\prec^+$ is chosen such that the first occurrence in l is the task t. For formalizing these advancing the advance counter, we introduce the macros $IsLegalEvalOrder(l)$ specifying whether a list of occurrences is a legal evaluation order w.r.t. $\prec^+$ and a macro $IsTopSort(l, s)$ specifying whether a list of occurrences l is a topological order of a set of occurrences s w.r.t. $\prec^+$:

$IsLegalEvalOrder(l) \triangleq \forall i, j : \mathbb{N} \bullet i < j \Rightarrow \neg l_j \prec l_i$

$IsTopSort(l, s) \triangleq IsLegalEvalOrder(l) \wedge (\forall n : OCC \bullet n \in l \Leftrightarrow n \in s)$

Then, advancing the program counter is formalized by the macro:

```
Proceed ≙
    choose t : OCC • t ∈ pc.nt
        pc := t
        if EntersExpr(t) then
            choose l : [OCC] • IsTopSort(l, t.subExprsIn) ∧ hd(l) = t
                toBeEvaluated := tl(l)
            end choose
        end if
    end choose
```

When the program counter is advanced within an expression, then the next occurrence of the list of sub-expressions to be evaluated is chosen. However before this choice, the occurrences that must not be evaluated if the current expression evaluates to v have to be removed from this list. If the current expression is a root, then we advance to the next task as specified by the macro *Proceed*. Hence, the macro

```
ProceedInExpr(v) ≙
    if ¬isEmpty(toBeEvaluated) then
        pc := hd(remove(toBeEvaluated, NotEvaluated(pc, v))
        toBeEvaluated := tl(remove(toBeEvaluated, NotEvaluated(pc, v))
    else Proceed
    end if
```

formalizes the advancing of the program counter within expressions.

For formalizing programming language semantics, all transition rules for tasks which are not expressions use the macro *Proceed* for advancing the program counter. The transition rules for tasks which are expressions use the macro $ProceedInExpr(v)$ with the suitable value v.

Example 2. We formalize the simple assignment-operator and the logical and-operator of C:

if pc **is** *ASSIGN* **then**
  $mem(addr(pc.dest, env) := value(pc.src)$
  $value(pc) := value(pc.src)$
  $ProceedInExpr(value(pc.src))$
endif

if pc **is** *CONJ* **then**
  if $value(pc.lop) = 0$ **then**
    $value(pc) := 0$
    $ProceedInExpr(0)$
  else
    $value(pc) := value(pc.rop)$
    $ProceedInExpr(value(pc.rop))$
  endif
endif

The transition rule for assignments uses a dynamic function $addr : OCC \times ENV \rightarrow ADDRESS$ for storing results of address computations (analogously to the dynamic function *val*). In addition to transition rules for the tasks of a programming language, it is only necessary to specify the order $\prec$ and the predicate $WillBeEvaluated(o, v, o')$.

7 Examples

In this section, we demonstrate the power of our approach by using this framework for the languages Ada, C, C++, Java, C#, and Fortran.

7.1 Ada

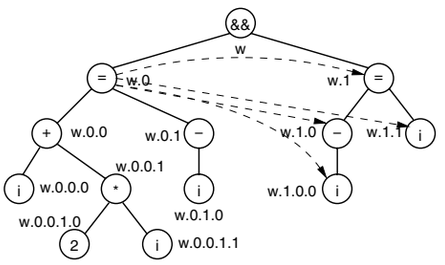


Fig. 6. Evaluation Order in Ada

Ada[10] defines fully non-deterministic expression evaluation except special boolean short-circuit operators **andthen** and **orelse**. In general, fully non-deterministic expression evaluation only requires that operands are evaluated before their operators. However, if an operand w is a left operand of short circuit operator, then it must be evaluated before any sub-expression of the corresponding right operand. Their occurrences have the common prefix $parent(w).1$. Hence, $\prec$ is defined as follows:

$$w \prec w' \triangleq w' = parent(w) \vee (parent(w) \textbf{ is } SHORTEVAL) \wedge isPrefix(parent(w).1, w').$$

The dashed arrows in Figure 6 show the precedences required for short circuit evaluation. It remains to define the predicate $WillBeEvaluated(o, v, o')$. The occurrence o' only will not be evaluated if one of the following cases occur:

- o is left operand of an operator **andthen**, $v =$ **false**, and o' belongs to the corresponding right operand.
- o is left operand of an operator **orelse**, $v =$ **true**, and o' belongs to the corresponding right operand.

Hence, the following macros formalize the predicate $WillBeEvaluated(o, v, o')$:

$$\begin{aligned}
&WillBeEvaluated(o, v, o') \triangleq \neg AndThen(o, v, o') \wedge \neg OrElse(o, v, o')\\
&AndThen(o, v, o') \triangleq (parent(o) \textbf{ is } ANDTHEN) \wedge o = parent(o).0 \wedge\\
&\qquad\qquad v = \mathsf{false} \wedge isPrefix(parent(o).1, o')\\
&OrElse(o, v, o') \triangleq (parent(o) \textbf{ is } ORELSE)o = parent(o).0 \wedge\\
&\qquad\qquad v = \mathsf{true} \wedge isPrefix(parent(o).1, o')
\end{aligned}$$

7.2 C and C++

We already discussed the informal language semantics of C in Section 4. C++ [12] is similar except that assignments define lvalues as the lvalues of their destinations. Therefore, C++ allows for example expressions like $(x = y) = z$ while these are forbidden in C. Otherwise, C++ defines the expression evaluation as C. Therefore, the predicates $\prec$ and $WillBeEvaluated(o, v, o')$ are defined equally for both languages. C requires that the operands are evaluated before their operators with one exception: if the operator is an assignment operator it needs only be executed before the next sequence point. Hence, an occurrence w is executed before another occurrence w' only if one of the following

- $w' = parent(w)$ and neither w nor w' is an assignment operator,
- w is not an assignment operator and w is left operand of w'
- w is the source of an assignment operator and w' is prefix of w.

These properties are formalized by:

$$\begin{aligned}
Basic(w, w') \triangleq\ &\neg(w \textbf{ is } ASSIGN) \wedge \neg(w' \textbf{ is } ASSIGN) \vee\\
&isPrefix(w', w) \wedge isSrc(w) \vee\\
&\neg(w \textbf{ is } ASSIGN) \wedge w'.0 = w
\end{aligned}$$

The dashed arrows with filled heads show these dependencies in the example of Fig. 7. The logical and-operator and the logical or-operator follow short-circuit evaluation and require that the left operand is evaluated before their right operand. Similarly, the comma operator requires that the left operand is evaluated before the right operand, and the conditional expression requires that first operand is evaluated before theo other two operands. This is formalized analogously to Ada:

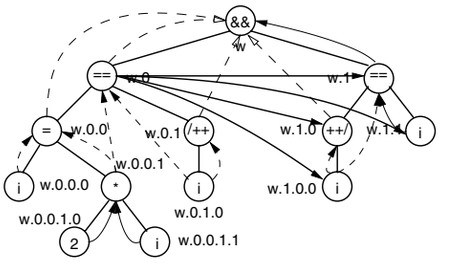


Fig. 7. Evaluation Order of Expressions in C and C++

$$\begin{aligned}
&LeftToRight(w, w') \triangleq\\
&(parent(w) \textbf{ is } SHORTEVAL) \wedge isPrefix(parent(w).1, w') \vee\\
&(parent(w) \textbf{ is } COND) \wedge isPrefix(parent(w).1, w') \wedge isPrefix(parent(w).2, w')) \vee\\
&(parent(w) \textbf{ is } COMMA) \wedge isPrefix(parent(w).1, w')
\end{aligned}$$

Here, *SHORTEVAL* is the common super-sort of *COND* and *DISJ*. The dependencies defined by *LeftToRight* are illustrated by the solid arrows in the example of Fig. 7.

Finally, any assignment must be completed before the sequence points. For simplicity, we require that any sub-expression of a sequence point must be evaluated before the sequence point. If the sequence point has an occurrence w', then the occurrences of all sub-expressions of w' have the prefix w'. There are sequence points that can explicitly be specified by their sort such as the comma operator or function calls, and there are sequence points that are specified as the first operand of short-circuit operator or conditional expression. Hence, this evaluation order is specified by

$$\begin{aligned} \mathit{SeqPoint}(w, w') \triangleq\ & (w' \textbf{ is } \mathit{SHORTEVAL}) \wedge \mathit{isPrefix}(w'.0, w) \vee \\ & (w' \textbf{ is } \mathit{COND}) \wedge \mathit{isPrefix}(w'.0, w) \vee \\ & (w \textbf{ is } \mathit{SEQPOINT}) \wedge \mathit{isPrefix}(w', w) \end{aligned}$$

The sort *SEQPOINT* is the common super-sort of the sort of function calls *CALL*, the sort of comma operators *COMMA*, and the sort of expression statements *EXPRSTAT*. The dependencies defined by *SeqPoint* are illustrated by the dashed arrows with empty heads in the example of Fig. 7. Thus, $w \prec w' \triangleq \mathit{Basic}(w, w') \vee \mathit{LeftToRight}(w, w') \vee \mathit{SeqPoint}(w, w')$ formalizes the evaluation order of C and C++. For the short-circuit logical operators, the predicate $\mathit{WillBeEvaluated}(o, v, o')$ is defined as in Ada. In addition, it must be specified that the third operand of a conditional operator is not being executed if the condition evaluates to a value different from 0 and the second operand of a conditional operator is not being executed if the condition evaluates to 0. Hence:

$$\begin{aligned} & \mathit{WillBeEvaluated}(o, v, o') \triangleq \neg \mathit{And}(o, v, o') \wedge \neg \mathit{Or}(o, v, o') \wedge \neg \mathit{Cond}(o, v, o') \\ & \mathit{And}(o, v, o') \triangleq (\mathit{parent}(o) \textbf{ is } \mathit{CONJ}) \wedge v = 0 \wedge \mathit{isPrefix}(\mathit{parent}(o).1, o') \\ & \mathit{Or}(o, v, o') \triangleq (\mathit{parent}(o) \textbf{ is } \mathit{DISJ}) \wedge v \neq 0 \wedge \mathit{isPrefix}(\mathit{parent}(o).1, o') \\ & \mathit{Cond}(o, v, o') \triangleq (\mathit{parent}(o) \textbf{ is } \mathit{COND}) \wedge v \neq 0 \wedge \mathit{isPrefix}(\mathit{parent}(o).2, o') \vee \\ & \qquad\qquad (\mathit{parent}(o) \textbf{ is } \mathit{COND}) \wedge v = 0 \wedge \mathit{isPrefix}(\mathit{parent}(o).1, o') \end{aligned}$$

7.3 Java and C#

Java[3] and C# [2] require a strict left-to-right evaluation order. As for the other languages, operands of an operator must be evaluated before the operator is evaluated. If two operands have the same parent (i.e. they are siblings), then this operands are evaluated from left to right, i.e. they are ordered according to the last number in their occurrence. Thus, the evaluation order is specified by:

$$w \prec w' \triangleq w' = \mathit{parent}(w) \vee (\mathit{parent}(w') = \mathit{parent}(w) \wedge (\mathit{last}(w) < \mathit{last}(w')))$$

Since Java and C# have the same operators as C and define the logical and-operator and the logical or-operator by short circuit evaluation, the predicate $\mathit{WillBeEvaluated}(o, v, o')$ is defined as for C.

7.4 Fortran

Fortran [11] defines fully non-deterministic expression evaluation order without short-circuit evaluation for boolean operators. Furthermore, Fortran has no conditional expressions. Hence,

$w \prec w' \triangleq w' = \mathit{parent}(w)$ and $\mathit{WillBeEvaluated}(o, v, o') \triangleq \mathsf{true}$

However, Fortran additionally specifies that the value of an expression does not depend on the order of its evaluation (although the state might be different). This is a property on ASM-runs and can therefore not be formalized by means of abstract state machines. Since ASMs define transition systems, temporal logic might be used for defining such requirements.

8 Conclusions

We developed a framework for specifying the dynamic semantics of expression evaluation using ASMs. Based on a predicate $\prec$ defining the evaluation order and on a predicate *WillBeEvaluated* defining suppression of evaluation of subexpressions, it is possible to define how the program counter advances. This definition is independent of the concrete language. For a concrete language, only these two predicates have to be defined (in addition to the other state changes caused by the execution of the operators). We demonstrated our approach by specifying these predicates for several real-life languages.

The kind of expression evaluation semantics as shown in this article is inherently non-compositional in its syntactic structure. Hence a straightforward definition of inference rules based on the program structure seems impossible. However, the formalization of non-deterministic expression evaluation by inference rules could follow a similar approach as in this paper – even for a big-step semantics. On the other hand, in special cases (for instance in the absence of side-effects), the semantics may be compositional. With an ASM-semantics, it is possible to prove the soundness of such inference rules. In general, an SOS can be proven as sound and complete w.r.t. an ASM semantics. Hence, the combination of both worlds allows to take the best of the two worlds for different purposes. Typical compiler correctness proofs are based on simulation proofs that could be formalized by means of inference rules. In our experience, inference rules seem more suitable as a basis for (mechanical) formal verification while ASM rules seem more suitable as a basis for validation of informal language specifications (in particular those for realistic languages as mentioned in the introduction).

A possible application of our approach could also be in the field of modern intermediate languages based on SSA-forms (static single assignment). SSA-forms only represent data-flow. There are so-called ϕ-nodes representing conditional data-flow. The evaluation of SSA-forms needs only be compatible with data-flow – the same requirement as for non-deterministic expression evaluation. Therefore, the same principles can be applied. However, the data-flow may contain loops, and it is not obvious how our approach can be extended taking into account data-flow graphs with cycles.

Finally this work could be considered as a starting point for providing frameworks for “dirty” language semantics as they are usual in real-life languages. Certainly, the dynamic semantics of expression evaluation is one of the most weird features, especially for C and C++. However, other features such as return statements, break statements and continue statements are also non-compositional in its nature. They could be formalized by continuations. However, these become

complicated, if loops are named, and it is possible to break or continue a loop with a name. This is not artificial but e.g. possible in Ada (w.r.t break) and Fortran (w.r.t. break and continue).

Acknowledgements

This work is supported by the DFG-project *Verifix*. We are grateful to our colleagues from the *Verifix*-project for their inspiring discussions that lead to this paper. We thank Egon Börger, who suggested to generalize a previous version of this work (formalizing C expression evaluation) to a framework that demonstrates differences and similarities of the semantics of expression evaluation for different programming languages.

A Operations on Lists, Occurrences, and Sets

The sort of sets over a sort T is denoted by $\{T\}$. Sets are denoted as usual and have the usual operations ($\cup$, $\cap$, $\setminus$, $\in$). We also use set comprehensions. For instance, if $\varphi(x)$ is a logical formula with free variable x of sort S, the set $\{x \bullet \varphi(x)\}$ denotes the set of all elements of the carrier set of S such that $\varphi(x)$ is satisfied. The sort of lists over a sort T is denoted by $[T]$. $hd(l)$ denotes the first element of a list l (the *head* of l) and $tl(l)$ denotes the list that is obtained from list l by removing its first element (the *tail* of l). $last(l)$ denotes the last element of a list. These operations are only defined for non-empty lists. l_i denotes the i-th element of a list l (l_0 is the first element). $|l|$ denotes the number of elements in list l. Observe that l_i is defined iff $0 \leq i < |l|$. The predicate $x \in l$ is satisfied iff x occurs in l, i.e., there is an i such that $l_i = x$. The predicate $isPrefix(l, l')$ is satisfied iff list l is a prefix of list l', i.e. it is $l_i = l'_i$ for all i, $0 \leq i < |l|$.

Occurrences are lists of natural numbers. ε denotes the empty occurrence. $o.n$ denotes the occurrence obtained by appending at the end a natural number n to occurrence o. $parent(o)$ defines the parent of an occurrence o, i.e. it is the occurrence obtained from o by removing its last element.

References

[1] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science*, pages 353–404. Springer, 1998. 398

[2] Microsoft Corporation. *C# Language Specification*. Microsoft Press, 2001. 392, 403

[3] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996. 392, 403

[4] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995. 393

[5] Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan EECS Department, 1997. 393
[6] Y. Gurevich and J. K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, volume 702 of *Lecture Notes in Computer Science*, pages 274–308. Springer, 1993. 398
[7] J. K. Huggins and W. Shen. The static and dynamic semantics of C. Technical Report CPSC-2000-4, Computer Science Program, Kettering University, 1999. 392, 398
[8] P. Kutter and A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997. 398
[9] M. Norrish. *C formalized in HOL*. PhD thesis, University of Nottingham, 1998. 397
[10] International Standardization Organization. International standard: Programming languages – Ada, 1995. ISO/IEC 8652:1995. 392, 401
[11] International Standardization Organization. International standard: Programming languages – Fortran, part 1: Base language, 1997. ISO/IEC 1539-1:1997. 392, 403
[12] International Standardization Organization. International standard: Programming languages – C++, 1998. ISO/IEC 14822:1998. 391, 392, 402
[13] International Standardization Organization. International standard: Programming languages – C, 1999. ISO/IEC 9899:1999. 392, 396
[14] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001. 392, 398
[15] C. Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methodes*, pages 131–164. Oxford University Press, 1995. 398
[16] C. Wallace. The semantics of the Java programming language. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, 1997. 392

Using AsmL for Runtime Verification

Mike Barnett, Wolfram Schulte, and Nikolai Tillmann

Foundations of Software Engineering, Microsoft Research
Redmond, Washington 98052-6399, USA
{mbarnett,schulte,nikolait}@microsoft.com

We use AsmL, an executable specification language created by the Foundations of Software Engineering group [2], to describe the behavior of .NET components and perform runtime verification. AsmL is based on the theory of ASMs; AsmL is an industrial-strength specification language that is a full member of the Microsoft .NET Framework.

Components are described at the API level by *rich interfaces*. A rich interface contains not only the syntax of an interface, but also its semantics via ASM transitions for each method operating on a set of instance fields.

Runtime verification dynamically monitors the behavior of a component; any discrepancy in the implementation's behavior relative to its specification is detected. AsmL specifications are compiled into a special format and then attached directly to the IL of an implementation component. Conceptually, the specification is run in lock-step with the implementation. At each method boundary, the results of the implementation are checked to see if they are allowed by the specification. We are able to track not just simple atomic data types, such as integers and strings, but also complex object and interface references. In addition, a rich interface can describe the interaction patterns that a component must be observed to engage in. These *mandatory calls* are the inter-component method calls that a component must be observed to make as part of its behavior. Full details are describe in [1].

References

[1] Mike Barnett and Wolfram Schulte. Contracts, components and their runtime verification on the .NET platform. Technical Report MSR-TR-2002-38, Microsoft Research, April 2002. Available from http://research.microsoft.com/pubs. 407

[2] Microsoft Research Foundations of Software Engineering, 2002. http://research.microsoft.com/fse. 407

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 407, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Modeling Information Services on the Basis of ASM Semantics

Aleksander Binemann-Zdanowicz and Bernhard Thalheim

Computer Science Institute, Brandenburg University of Technology at Cottbus
PostBox 101344, D-03013 Cottbus, Germany
{binemann,thalheim}@informatik.tu-cottbus.de

Abstract. Operational semantics for database systems is still an open research area. Moreover, the development of appropriate operational semantics for challenging applications becomes crucial and indispensible. Such applications are, for instance, internet applications which can be modeled on the basis of SiteLang.

1 Introduction

Information systems have become an integral component of current computer infrastructure. Due to the growing complexity of applications, the development of appropriate and satisfying information systems is still a challenging task. Information systems technology has evolved to a very large extent. Methodology for development of large information-intensive systems has not evolved in the same speed. We also observe an impedance mismatch between software engineering methodologies and database development methodologies.

Typical for the development of internet services is the brute-force programming approach. Web sites are developed by teams without using design methodologies, are under constant change and thus quickly become inconsistent, of low usability and are not maintainable.

This paper aims in developing a method that allows to check an information service before it is produced and put into practice. We use a specification language (SiteLang) for consistent development of services, providing operational semantics based on the ASM approach, which is invariant throughout the entire development cycle and allows to show a number of application properties and enables the generation of executables.

2 Challenges of Applications

The motivation for SiteLang comes from our website projects and, particularly, from the development of the interactive platform *Cottbus interaktiv*, enabling viewers of TV stations to send requests for personalized video/radio on demand, internet on user profile, user-sensitive electronic program guide etc. over a TV channel. The respective personalized services and contents are then provided onto their set-top-boxes.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 408–410, 2003.
© Springer-Verlag Berlin Heidelberg 2003

The main problem caused by the brute-force programming approach chosen for *Cottbus interaktiv* is the uncertainty whether the code survives the next hardware and/or software update. The development of *Cottbus interaktiv* had to be pursued under not yet stable technology standards nor software environments. Thus, we have an urgent need of an operational semantics for information service specifications.

3 The SiteLang Approach

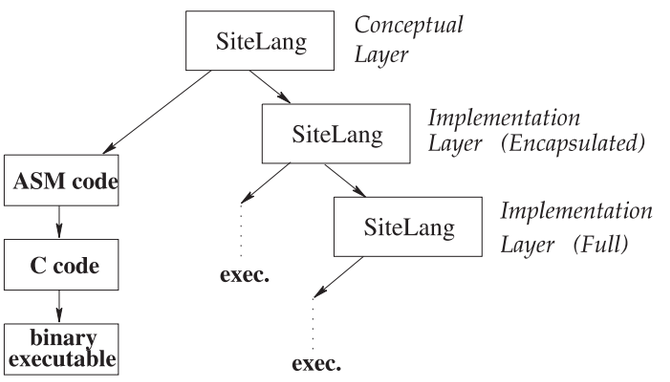
The SiteLang language comprises constructs for specifying *database functionality* (database schema, database operations, transaction management, integrity constraints, database content), as well as *user interaction* (event-driven interaction model, multiple users and devices, scenes, dialog steps with transaction semantics, dialogs, media objects) [DuT01]. It also supports specifying distributed system architectures.

The SiteLang approach is based on the theoretical foundations of the CoDesign work [DuT01] and of database theory. It is platform-independent and provides abstract support for modeling different database systems, heterogenous system components, clients with varying user interaction capabilities etc.

4 Main Results

Our work provides an ASM-based definition of operational semantics for information services. It contains a proof-of-concept solution developed for and used in SiteLang and especially for specifying interactive television services. An important feature is the stability of semantics of information services during specification refinement and in changing environments.

The semantics of SiteLang is defined by means of the corresponding ASM code, which is generated automatically on any specification refinement level and translated to an executable binary file by the *Xasm* compiler, as depicted below.



References

[Anl00] M. Anlauff: *An Extensible, Component-Based ASM Language*, in: *ASM'2000*, LNCS 1912, Springer, 2000, pp. 69-90

[Bin01] A. Binemann-Zdanowicz: *Towards Information System Modeling on the Basis of ASM Semantics*, BTU Cottbus, Preprint I-12/2001

[DuT01] A. Düsterhöft, B. Thalheim: *Conceptual modeling of internet sites*, in: *ER'2001*, LNCS 2224, Springer, 2001, pp. 179-192 409

Designing the Parlay Call-Control Using ASMs

Paolo Falcarin[1] and Alessandra Cavarra[2]

[1] Dip. Automatica e Informatica, Politecnico di Torino
paolo.falcarin@polito.it
[2] Oxford University Computing Laboratory
ale@comlab.ox.ac.uk

Recently different standardization bodies have joined to create a new standard to open network API between the telecom service level, which gives connectivity to different networks, and the application level, which is built by enterprises outside the telecom network domain. This OSA (Open Service Access) API is a standard interface for applications to use the capabilities of a network without knowing technology details. Parlay/OSA [4] specification covers different aspects: messaging, location, conferencing, security, call-control, etc. Our work focuses on the Call Control API specification whose main goal is both to allow applications creating whichever kind of call between voice terminals connected to different networks, and also react to call requests coming from various networks, according to the service-logic implemented by the applications (e.g. [3]). The Parlay documentation provides a specification of the Call Control Service Capability Feature (SCF) aspects of the interface by means of UML [5] class, state, and sequence diagrams. However, although UML emerges as the best practice modeling notation for object-oriented software development, in its current setting it still lacks of a formal semantics. As a consequence its usage and, correspondingly, the interpretation of developed models may differ considerably. For this reason several of semantics models for UML diagrams have been provided (see for instance [1]). We present a methodology to produce an ASM specification of the Parlay call control from the UML diagrams provided in the documentation. We exploit an existing UML to ASM tool framework [2] able to obtain *automatically* an ASM model of the Parlay call control given through UML class and state diagrams. This approach is of interest to groups working on Parlay mainly for two reasons. (a) **Validation**. In the Parlay specification a number of UML sequence diagrams modeling important scenarios of application are given. Feeding our toolkit with these sequence diagrams, such scenarios can be simulated against the given model to check that the system conforms with the expected behavior. (b) **Maintainability**. The update of a protocol specification, the supply of an additional protocol and of new scenarios are a big problem for developers because any modification in this complex software implies a new series of verifications of standard compliance, and leads to new potential bugs. Changes in the Parlay UML specification would be reflected in the corresponding ASM model automatically created by the tool. Additional effort would only be required to change the ASM specification of actions and events.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 411–412, 2003.
© Springer-Verlag Berlin Heidelberg 2003

References

[1] E. Börger, A. Cavarra, and E. Riccobene. A precise semantics of UML State Machines: making se- mantic variation points and ambiguities explicit. In Proc. Semantic Foundations of Engineering Design Languages (SFEDL02) - ETAPS 2002. 411

[2] A. Cavarra, E. Riccobene, and P. Scandurra. A Framework to Simulate UML Models. Submitted to TACAS 2003. 411

[3] P. Falcarin and P. Lago. Call Control Component implementing converged Telephony-Internet networks. In NCC, co-located with ICSE 2001, 2001. 411

[4] Parlay specification 3. http://www.parlay.org. 411

[5] OMG Unified Modeling Languages Specification, version 1.4, 2001. 411

Test Case Generation from AsmL Specifications

Wolfgang Grieskamp, Lev Nachmanson, Nikolai Tillmann, and Margus Veanes

Microsoft Research
One Microsoft Way, Redmond, WA 98052
{wrwg,levnach,nikolait,margus}@microsoft.com

Model-based testing has recently been receiving more and more attention in Microsoft's product groups. Typically, the models are required to be finite state machines (FSMs) because there is well-known algorithmic support for test case generation for FSMs. The direct FSM based modeling approach has its limitations though. Most noticeably, the FSM model for a particular testing problem may not be the most natural one to describe the implementation under test. This motivated us to study the problem of algorithmic generation of an FSM from a given AsmL spec.[1] We have developed a pragmatic test case generation tool where the FSM generation algorithm is used and has been extended in several ways.

Given an AsmL spec *S*, the first step of the user of the tool is to identify the following meta information about *S*: *input actions, observed state variables*, *test properties* and *filters*. *Input actions* are named rules (methods) of *S* together with *parameters*. The tool can infer valid parameter values by means of efficient pruning of their domains through user-defined constraints. Input actions can be individually fired in the model and constitute the atomic building blocks of test cases. *Observed state variables* form a subset of the state variables of *S*; the user may deliberately want to ignore some variables from the state exploration point of view by not including them among the set of observed variables. *Test properties* are finite-valued expressions (eg. Boolean expressions) for grouping states together. *Filters* are Boolean expressions that are used to restrict the state space by filtering out undesired states. Collectively, the role of test properties and filters is to bound the explored state space during the FSM generation.

The FSM-generation algorithm is a particular kind of graph reachability algorithm. It starts from the initial state of the ASM and builds up a state transition graph by executing the actions. Intuitively, it produces a finite unwinding of the (typically) infinite transition system that is implicitly described by the ASM. The FSM generation algorithm uses built-in meta-programming support available in AsmL runtime for exhaustive exploration of the state space. To fully explore nondetrminism a depth-first search over all internal choices of each action is conducted using the *explore* construct of AsmL.

The FSM generation algorithm produces a finite approximation of the ASM. View the FSM as a directed graph and mark each link with the cost of executing the corresponding action at the corresponding state. The tool can be used to generate test cases (in form of action sequences that start from the initial state) that provide full coverage of the links of the graph for the minimal total cost.

---

[1] W. Grieskamp, Y. Gurevich, W. Schulte and M. Veanes, Generating Finite State Machines from Abstract State Machines, ISSTA 02, Software Engineering Notes 27(4) 112-122, ACM, 2002.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 413, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Teaching ASMs, Teaching with ASMs: Opportunities in Undergraduate Education

James Huggins[1], Jean Mayo[2], and Charles Wallace[2]

[1] Computer Science Program, Kettering University
Flint, Michigan, USA
[2] Computer Science Department, Michigan Technological University
Houghton, Michigan, USA

Do people truly find ASMs easy to understand and use? As educators, we are particularly interested in how readily this new technology transfers to the classroom. Furthermore, we would like to take it beyond courses like software engineering, where formal methods are commonly encountered, and extend it to other areas. As many experts have pointed out, the only way in which advanced software engineering tools and methods can make headway into popular software design is if they are woven into the fabric of traditional computer science courses. We report on our current work in bringing ASMs to the classroom.

Instructional materials for ASMs. In the tradition of Gurevich's original ASM tutorial and subsequent works, our *ASM primer* is a dialogue between the authors and an inquisitive undergraduate student, Questor. This style adds a certain degree of levity to what can be a rather dry exposition. It also allows us to empathize with the student readers, addressing issues they typically find difficult as they arise. From experience, we know where the obstacles to understanding lie: the notion of functions that change meaning, the parallelism of block rules, the representation of complex computations simply as functions.

ASMs in a software engineering curriculum. As part of the new Software Engineering degree option, a new course has been added, focussing on aspects of the software process concerned with quality assurance: requirements, specification, formal description and modeling, and verification The course starts with an investigation of "problem structuring and analysis" through *problem frames*, which provides a good foundation for the subsequent coverage of ASMs.

ASMs in a concurrent computing curriculum. We intend to promote the use of abstraction, information hiding, and refinement in the design of concurrent software by incorporating ASMs into the teaching of concurrent computing. We will provide *automated tool support* by incorporating the *XASM* tool into existing *ConcurrentMentor (CM)* visualization software. Students will be able to write software partly in C++, using the class libraries of CM, and partly in ASM. The ASM portions will be compiled to C++, using a modified version of XASM's ASM-to-C compiler. CM provides a C++ library of communication primitives and visualization capabilities. Students will benefit from the expressivity of ASM and the visualization and analysis features of CM. The movement from abstract design to concrete implementation will be achieved by gradually replacing ASM code with equivalent C++ code. A corpus of *documented examples* will also be created.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 414, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Using ASM Specifications for Compiler Testing

Alexey Kalinov, Alexandre Kossatchev, Alexander Petrenko, Mikhail Posypkin, and Vladimir Shishkov

Institute for System Programming of Russian Academy of Sciencies
{ka,kos,petrenko,posypkin,vova}@ispras.ru

Developing an adequate set of tests also called a **test suite** is an important part of software development process. We faced this problem while working on mpC parallel programming language compiler. The general task was to develop a test suite for checking whether the particular compiler implementation correctly processes the programming language.

In this case study we focus on testing language expressions. mpC provides powerful operators for array-based and parallel computations. That is why mpC expressions are complicated and difficult to implement and require thorough testing. However the proposed techniques is also applicable to other parts of the language.

Our approach is a sort of "specification-based testing". We use Abstract State Machines formalism for modeling mpC expressions semantics. The formal specification is implemented using the ASM-based Montages framework – a new method for giving the semantics of a programming language.

We use the specification for three different purposes:

- **Test cases generating.** mpC specification consists of several Montages. Each montage defines the semantics of a particular abstract syntax tree node. Test programs are generated by combining abstract syntax tree nodes. Incorrect tests are filtered out by the specification. Correct tests constitute the test suite.
- **Test oracle generating.** Executable specification is used for generating trustable output of the given test program. Test oracle compares actual and trustable outputs for a particular test. If results are not identical the verdict is failure.
- **Providing test coverage criteria.** Analysis of the specification coverage allows one to see whether all specification rules were involved while executing the test suite. If the coverage criteria are satisfied then no more test cases are needed, otherwise additional test programs should be added to the test suite.

The test suite of more then 13000 tests was generated with using technique. It has helped to find out a lot of bugs in the mpC compiler and significantly improve its quality.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 415, 2003.
© Springer-Verlag Berlin Heidelberg 2003

ASMs as Integration Platform towards Verification and Validation of Distributed Production Control Systems at Multiple Levels of Abstraction

Martin Kardos and Ulrich Nickel

University of Paderborn
Department of Computer Science and Mathematics, 33102 Paderborn, Germany
{kardos,duke}@uni-paderborn.de

Production systems used nowadays in manufacturing processes can be characterized as stationary, inflexible, production-specific systems with a strongly centralized control. Once such a production system has been built, it is very time and cost demanding to adapt it to new changes in the manufacturing process. On the contrary, the market competition and customer needs impose new requirements on the production systems. There are increasing demands for shorter production times, higher flexibility, reconfigurability and optimal utilization of production facilities. Such demands can be met only by distributed production systems (DPS) with a decentralized organization.

We introduce here a methodology[1] developed for the integrated design, analysis and validation of *distributed production control systems* (DPCS), i.e. the control part of the DPS. The defined methodology consists of several consecutive design steps, which together form a consistent development process. This covers all aspects of system development, i.e. analysis, specification, design and implementation. The methodology also aims on reuse of techniques and languages that are well accepted in software engineering. Based on the analysis of our reference *Manufacturing Flow System* case study, subsets of UML and SDL languages have been selected to form a modelling platform in our methodology. In order to support verification and validation by means of model-checking and simulation, a concept of *ASM meta-modelling* has been defined. This concept integrates both languages within one common formal semantic basis using the ASM formalism. The meta-model has been implemented in the ASM language (AsmL), thus giving us the possibility to validate its correctness by execution. Moreover, the meta-model supports a clear separation of semantic concerns such as asynchronous communication, state machine based control and OO-based action execution. This way, we obtain different abstraction levels at which a system model can be verified by means of model-checking.

Technically, the integration of model-checking into our methodology requires a transformation from the ASM meta-model, expressed in AsmL, into the input language of some model-checker like SMV or PROMELA. The definition of this transformation is still ongoing work.

[1] This work is part of the ISILEIT project supported by the German Research Foundation (SPP1064, GA 456/7).

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 416, 2003.
© Springer-Verlag Berlin Heidelberg 2003

AsmL Specification of a Ptolemy II Scheduler

Daniel Lázaro Cuadrado[1], Peter Koch[1], and Anders P. Ravn[2]

[1] Embedded Systems Group*, Center For PersonKommunication, Aalborg University
Niels Jernes Vej 12, DK-9220 Aalborg, Denmark
{kapokasa,pk}@kom.auc.dk
[2] Department of Computer Science, Aalborg University

Ptolemy II [1] is a Java implemented modeling, simulation and code generation framework for heterogeneous, concurrent, real-time embedded systems. A key issue in the framework is correct and efficient execution of heterogeneous models of computation within a common simulator by using the object-oriented principles of polymorphism and information hiding.

Component behavior is given by the programs and computational models underlying the simulator, so they must form the basis for reasoning about them. Yet, Java programs are not well suited as a basis for reasoning because Java is primarily a programming language; was not designed as a notation for giving semantics to a system. Therefore, there is a need for precise mathematical semantics. However, having mathematical semantics on paper, separate from an implementation, is not attractive because one would have to continually update it as the simulator evolves. Furthermore, one would still have to argue that there is a clear correspondence between the simulator program and the semantic definitions. An AsmL implementation of Ptolemy components would on one hand allow simulations and on the other hand allow reasoning about behavior. We present initial steps towards an AsmL implementation of the scheduler of the Synchronous Dataflow (SDF) domain in Ptolemy. SDF is a popular specification formalism for embedded software and hardware.

The derived formal specification of the SDF scheduler has proceeded as a reverse engineering phase, followed by analysis of the source code. Once we had enough knowledge about the software architecture as well as the functionality implemented, we initiated the implementation of the scheduler in AsmL. It was decided to stay close to the current implementation structure. This is important in order to have a realistic model of the current implementation, and to be prepared for possible further modifications of either the simulation tool or the AsmL model.

So far, the implementation includes mainly architectural content. A few basic functionalities have been included in different classes. The AsmL code is more than 3 times smaller than the Java implementation, which indicates the degree of abstraction achieved. In order to ease simulation of Ptolemy models, we have built a loader that takes models in an XML-format (called MoML, Modeling Markup Language) and loads them onto the AsmL platform. Initial experiments show that our implementation is just as fast as the Java implementation, although we have a more abstract description. This is probably due to the fact that ASML generates C++ code, which is not interpreted like Java.

---

* Department of EECS, UC Berkeley: *Ptolemy Project*. http://ptolemy.eecs.berkeley.edu/

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 417, 2003.
© Springer-Verlag Berlin Heidelberg 2003

ASM Specification of Database Systems

Andreas Prinz[1] and Bernhard Thalheim[2]

[1] DResearch Digital Media Systems GmbH
Otto-Schmirgal-Str. 3, D-10319 Berlin, Germany
prinz@DResearch.de
[2] Computer Science Institute, Brandenburg University of Technology at Cottbus
PostBox 101344, 03013 Cottbus, Germany
thalheim@informatik.tu-cottbus.de

Abstract. Database semantics is often reduced to semantics of static databases. So far, there is no theory of database dynamics or operational semantics for database applications. This paper aims in developing such a theory based on the ASM approach. We restrict the consideration to the business user layer.

1 State-Of-The-Art of Database Theory

Database theory is well developed as long as structuring of the relational database models is considered. The situation already changes in the case of structuring on the basis of relational structures and of relational integrity constraints. The situation becomes worse in the case of object-oriented database models and is much better for hierarchical entity-relationship model [Tha00] which are based on rigid hierarchical structuring.

Database dynamics is still open in the case of object-relational models. Concepts such as triggers, stored procedures and transactions are far from being well investigated. Wegner's interaction machines can be used for the development of a kernel theory of information systems [GST00]. Specific elements of interaction machines can be easily modeled [Bör01] on the basis of Abstract State Machines. This paper shows how a consistent and understandable theory of database systems can be developed and discusses operational semantics of database operating.

2 ASM Specification of Database Operation

Database systems are designed to operate in parallel. Operating of databases systems can be understood in a state-based approach. Our approach is based on different levels of abstraction:

Level 1 (Point of view of business users)**:** The database system defined by state spaces: The input state is based on algebraic structure with ground terms defined on the values and the names. The database state is based on the (object-)relational structure with well-defined composition operators. The output state is a general database defined on the values and names.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 418–420, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Level 2 (Conceptual point of view)**:** Database systems are defined as an extension of level 1 by transactions, constraints, views and integrity maintenance.

Level 3-1 (Logical point of view)**:** The logical database system defined as an extension of level 2 by states of the database management system by the transaction and recovery engine, by the synchronization engine, the logging engine and the query translating engine.

Level 3-2 (Physical point of view)**:** The physical database system is defined as an extension of level 3-1 by specific functions of the DBMS.

Level 4 (DBMS point of view)**:** On level 4, the storage engine is modeled in detail with the buffers and the access engine.

This separation allows to model database systems at different levels of detail. Each higher level should be a refinement of the lower levels.

3 ASM-Based Theory of Database Operating

We define a corresponding concrete database machine for each of these levels. They all use the same abstract interface, such we can challenge all of them with the same concrete database operations. The results should be the same in all the cases. We have chosen to challenge the specification by applying it to an existing application. The application chosen is the restaurant database system of http://www.cottbus.de. We could show sample requests derived by the ASM. The database systems specification has been now tested on the life input data of the application. The AsmL system was running and, thus, demonstrated the correctness of the approach. We used AsmL 2.0 to get an executable ASM specification.

Parallel runs of transactions must behave in the same manner as a sequential run of the transactions. This concept is entirely covered by partially ordered runs. Interaction of database systems with user or information systems can be handled by ASM. Database systems are considered on a variety of abstractions layers which is nieatly supported by the ASM approach.

4 Outlook and Future Work

So far, the first abstraction level and further a general theory of transactions have been developed. This last theory enhances the approach to database recovery provided by [GSW97]. Currently, a general treatment of level 2 is investigated.

References

[Bör01] E. Börger, Discrete Systems Modeling. Encyclopedia of Physical Sciences and Technology, Academic Press 2001. 418

[GST00] D. Goldin, S. Srinivasa and B. Thalheim, IS = DB+Interaction: Towards Principles of Information System Design. Proc. ER'2000 (eds. A. S. H. Laender, S. T. W. Liddle, V. C. Storey), LNCS 1920, Springer, Berlin, 140-153. 418

[GSW97] Y. Gurevich, N. Soparkar, and Charles Wallace, Formalizing database recovery. Journal of Universal Computer Science, 1997, 3, 4, 320-340. 419

[Tha00] B. Thalheim, Entity-relationship modeling – Foundations of database technology. Springer, Berlin, 2000. 418

The Computable Kernel of ASM

Wolfgang Reisig

Humboldt-Universität zu Berlin

A rich variety of system models for sequential, deterministic systems has been suggested during recent decades, including automata, process algebras, many versions of Petri Nets, and models to describe the semantics of Programming languages. All models for sequential, deterministic systems assume a set S of *states*, or *configurations*, and a next state function υ: $S \to S$. This yields *computation sequences* $S_0\ S_1\ S_2\ \ldots$ with $S_i = \upsilon(S_{i-1})$. One usually assumes a finite, symbolic representation of states. The next state function υ is usually computable on this representation.

ASM suggest a more liberal approach, where states S not necessarily have a finite, symbolic representation and – consequently – the next state function υ is not necessarily computable. This fits nicely with Don Knuth's «The art of computer programming». He starts out with *computational methods*: The next state function of a computational method « *...might involve operations that mortal man can not always perform*». Knuth defines *effective* computational methods as a special case where the next state function is essentially equivalent to a Turing Machine. In this framework, ASM are computational methods that are not effective in general.

Not every computational method is an ASM. First of all, a (sequential, bounded) ASM $\boldsymbol{M}$ is associated a *signature*, Σ. The set Alg_Σ of all Σ-algebras is the set of states of $\boldsymbol{M}$. The next state function of $\boldsymbol{M}$ is shaped

$$\mathrm{sem}_{\mathbf{M}}: \mathrm{Alg}_\Sigma \to \mathrm{Alg}_\Sigma .$$

This function is not computable, because its domain and its range are not countable; hence sem_M can not be encoded as a function over the natural numbers. Nevertheless, sem_M is « effectively structured». We outline this structure in the rest of this abstract.

Given an ASM $\boldsymbol{M}$, let $T_M \subseteq T_\Sigma$ be the set of Σ-groundterms that occur in the representation of $\boldsymbol{M}$. As a technicality, close T_M under subterms. Then each $R \in Alg_\Sigma$ can be associated a function φ_R: $T_M \to T_M$ such that for $S := sem_M(R)$ and for each $t \in T_M$ holds:

$$t_S = (\varphi_R(t))_R .$$

Hence, the interpretation of terms $t \in T_M$ in S can be expressed in R already, by help of φ_R . By induction on the structure of terms, φ_R can canonically be extended to all terms $t = f(t_1, \ldots, t_n) \in T_\Sigma \backslash T_M$, by

$$\varphi_R(t) := f(\varphi_R(t_1), \ldots, \varphi_R(t_n)) .$$

Then the above equality $t_S = (\varphi_R(t))_R$ extends to all $t \in T_\Sigma$. Furthermore, the extended function φ_R: $T_\Sigma \to T_\Sigma$ is computable, provided the equivalence $t_R = u_R$ is decidable in R for all terms $t, u \in T_\Sigma$.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 421-422, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Two states R and $S \in Alg_\Sigma$ are called *equivalent* iff for all t, u $\in T_M$ holds: $t_R = u_R$ iff $t_S = u_S$. The functions φ_R depend only on this equivalence: $\varphi_R = \varphi_S$ if R and S are equivalent. There exist only finitely many such equivalences (because T_M is finite); hence there is a number, k, and k functions φ_i: $T_\Sigma \to T_\Sigma$ (i = 1, … ,k) such that for all $R \in Alg_\Sigma$ the function φ_R is equal to some φ_i. It is now a technicality to assign each state $R \in Alg_\Sigma$ its index i and to integrate the functions φ_i into one function

$$\varphi : T_\Sigma \times \{1, \dots, k\} \to T_\Sigma \times \{1, \dots, k\} .$$

This function is the *kernel* of **M**.

Summing up, there is a computable function, φ, that for each step $R \to S$ of an ASM M describes the value of each term t $\in T_\Sigma$ in S, by help of terms and their values in R. Hence, for each function symbol f in Σ, the function f_S of S is characterized for all term generated argument tuples (u_1, … ,u_n) by help of R, φ, and the principle of induction. If one of the u_i is not be term generated, the value remains, i.e. $f_S(u_1, \dots, u_n) = f_R(u_1, \dots, u_n)$ in this case.

This completes the picture for steps R→S: Given a procedure to decide the equivalence of terms in R, the value $f_S(u_1, \dots, u_n)$ is given by a computable function on the finite term set T_M, or its inductive extension to T_Σ , or it is equal to $f_R(u_1, \dots, u_n)$.

A Non-standard Approach to Operational Semantics for Timed Systems

Heinrich Rust

Lehrstuhl Software-Systemtechnik, BTU Cottbus
Postfach 101344, D-03013 Cottbus, Germany
rust@informatik.tu-cottbus.de
Tel. +49(355)69-3801, Fax.: +49(355)69-3810

Operational semantics for linearly and quantitatively timed systems depend crucially on the model of time chosen. Two approaches are common: **(a)** Discrete models of time isomorphic to the natural numbers. Well established discrete formalisms can be used to describe the dynamical behavior of the system and time is uniform, but sequential refinement and composition of systems designed for different time scales are difficult, and continuous changes can not be well represented. **(b)** Hybrid models as sequences of intervals of the real numbers. Discrete events happen where such an interval ends (and, typically, the next one starts) and are assumed to take no time. Continuous changes take place during intervals. Point intervals are used to model consecutive discrete steps without an intervening continuous phase. Hybrid models can express the combination of continuous and discrete changes and allow sequential refinement of discrete steps. There are no problems for composition of systems because of differing bases of time. But this model of time is dense and non-uniform, and this makes reasoning more difficult.

In order to overcome the problems and to keep the nice points of both approaches, a novel discrete model of time using a constant strictly positive but infinitesimal step width is proposed, building on Nelson's axiomatic approach to infinitesimals. This model of time is isomorphic to the natural numbers, and thus, formalisms for the description of untimed discrete systems can be applied to the description of timed systems basically unchanged. We use Gurevich's Abstract State Machines as the discrete base formalism. On the other hand, between any two real numbers there is a sufficient supply of moments (in the language of non-standard analysis: not infinite, but unlimited), so as long as infinitesimal differences can be ignored, sequential refinement is not a problem; the set of points in time is dense in the real numbers.

Classical concepts like infinite and unbounded activity and receptivity are transferred to the new model, where they are simpler to formulate. Purely discrete means suffice for proving some properties of systems with continuously changing components. The flexibility of ASMs allows to model other real-time formalisms in our approach basically without coding. The concept of infinitesimal discretizability can be defined which allows to investigate if a basic condition is fulfilled for a hybrid algorithm to be implemented as a discrete system.

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, pp. 423–424, 2003.
© Springer-Verlag Berlin Heidelberg 2003

The main problems of the new approach are the impossibility of simulating a system step by step (for spanning any non-infinitesimal interval of time, an unlimited number of steps is needed), and the fact that the concepts of non-standard analysis are uncommon in computer science.

Parallelism versus Nondeterminism – On the Semantics of Abstract State Machines

Wolfram Schulte

Foundations of Software Engineering
Microsoft Research, Redmond, WA 98052, USA

Abstract State Machines (ASMs) are a simple but powerful formalism for modeling both hardware and software. Models typically allow an implementation freedom for a range of behavior. ASMs use two concepts to achieve this. Parallel composition abstracts from irrelevant sequentiality. Nondeterministic choice abstracts from irrelevant algorithmic detail. As a thought experiment, we wish to see the consequences of substituting parallel composition by nondeterministic composition. We have found that it provides some alternative benefits, e.g. the complex theory of partial updates wouldn't be necessary, since succesSsive updates take care of combining the effect of partial updates. ASMs of different abstraction levels could be reused, since updates on different abstraction levels wouldn't collide. It would be consistent with the current mechanism for exception handling in Microsoft's ASM language. Finally, we also show that this change in semantics does not destroy the spirit of ASMs, in fact most ASM models wouldn't need any change at all. We have not analyzed the consequences of using nondeterminstic composition for the ASM thesis, but for a specification language nondeterminstic composition seems to work fine. For a detailed analysis, see [1].

References

[1] Wolfram Schulte. Parallelism versus nondeterminism – On the semantics of abstract state machines. Technical Report MSR-TR-2002-109, Microsoft Research, November 2002. Available from `http://research.microsoft.com/pubs`. 425

E. Börger, A. Gargantini, E. Riccobene (Eds.): ASM 2003, LNCS 2589, p. 425, 2003.
© Springer-Verlag Berlin Heidelberg 2003

Author Index

Artho, Cyrille 87
Asmundo, Marianna Nicolosi ... 324
Awhad, Varsha 166

Barnett, Mike 407
Beierle, Christoph 186
Benczúr, András 204
Bertolino, Antonia 1
Binemann-Zdanowicz, Aleksander 408
Börger, Egon 22, 218
Boiten, Eerke 33
Bolognesi, Tommaso 22, 218

Cavarra, Alessandra 229, 411
Cuadrado, Daniel Lázaro 417

Derrick, John 33
Dold, Axel 391
Drusinksy, Doron 87

Falcarin, Paolo 411
Fruja, Nicu G. 244

Gargantini, Angelo 263
Gawanmeh, Amjad 278
Glässer, Uwe 204
Glesner, Sabine 293
Goldberg, Allen 87
Grädel, Erich 309
Grieskamp, Wolfgang 413
Gurevich, Yuri 57

Havelund, Klaus 87
Huggins, James 414

Kalinov, Alexey 415
Kardos, Martin 416
Kern-Isberner, Gabriele 186
Koch, Peter 417
Kossatchev, Alexandre 415

Lowry, Mike 87
Lukovszki, Tamás 204

Mayo, Jean 414
Meyer, Bertrand 108

Nachmanson, Lev 413
Nickel, Ulrich 416
Nowack, Antje 309, 341

Ober, Ileana 356

Pasareanu, Corina 87
Payton, Jamie 126
Petrenko, Alexander 415
Posypkin, Mikhail 415
Prinz, Andreas 418

Ravn, Anders P. 417
Reisig, Wolfgang 421
Riccobene, Elvinia ... 229, 263, 324
Rinzivillo, Salvatore 263
Roman, Gruia-Catalin 126
Rosenzweig, Dean 372
Roşu, Grigore 87
Runje, Davor 372
Rust, Heinrich 423

Scandurra, Patrizia 229
Schulte, Wolfram 407, 425
Shishkov, Vladimir 415
Slani, Neva 372
Stärk, Robert F. 244
Stevens, Perdita 151

Tahar, Sofiène 278
Thalheim, Bernhard 408, 418
Tillmann, Nikolai 57, 407, 413

Veanes, Margus 413
Visser, Willem 87

Wallace, Charles 166, 414
Winter, Kirsten 278

Zimmermann, Wolf 391