Lecture Notes in Computer Science 2621

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

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

Mauro Pezzè (Ed.)

Fundamental Approaches to Software Engineering

6th International Conference, FASE 2003
Held as Part of the Joint European Conferences
on Theory and Practice of Software, ETAPS 2003
Warsaw, Poland, April 7-11, 2003
Proceedings

Springer

Volume Editor

Mauro Pezzè
Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Via Bicocca degli Arcimboldi, 8, 20126 Milano, Italy
E-mail: pezze@disco.unimib.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 Nationalbibliografie; detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743
ISBN 3-540-00899-3 Springer-Verlag Berlin Heidelberg New York

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

http://www.springer.de

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin GmbH
Printed on acid-free paper SPIN: 10872970 06/3142 5 4 3 2 1 0

Foreword

ETAPS 2003 was the sixth instance of the European Joint Conferences on Theory and Practice of Software. ETAPS is an annual federated conference that was established in 1998 by combining a number of existing and new conferences. This year it comprised five conferences (FOSSACS, FASE, ESOP, CC, TACAS), 14 satellite workshops (AVIS, CMCS, COCV, FAMAS, Feyerabend, FICS, LDTA, RSKD, SC, TACoS, UniGra, USE, WITS, and WOOD), eight invited lectures (not including those that are specific to the satellite events), and several tutorials. We received a record number of submissions to the five conferences this year: over 500, making acceptance rates fall below 30% for every one of them. Congratulations to all the authors who made it to the final program! I hope that all the other authors still found a way of participating in this exciting event, and I hope you will continue submitting.

A special event was held to honor the 65th birthday of Prof. Wlad Turski, one of the pioneers of our young science. The deaths of some of our "fathers" in the summer of 2002 – Dahl, Dijkstra and Nygaard – reminded us that Software Science and Technology is, perhaps, no longer that young. Against this sobering background, it is a treat to celebrate one of our most prominent scientists and his lifetime of achievements. It gives me particular personal pleasure that we are able to do this for Wlad during my term as chairman of ETAPS.

The events that comprise ETAPS address various aspects of the system development process, including specification, design, implementation, analysis and improvement. The languages, methodologies and tools which support these activities are all well within its scope. Different blends of theory and practice are represented, with an inclination towards theory with a practical motivation on the one hand and soundly based practice on the other. Many of the issues involved in software design apply to systems in general, including hardware systems, and the emphasis on software is not intended to be exclusive.

ETAPS is a loose confederation in which each event retains its own identity, with a separate program committee and independent proceedings. Its format is open-ended, allowing it to grow and evolve as time goes by. Contributed talks and system demonstrations are in synchronized parallel sessions, with invited lectures in plenary sessions. Two of the invited lectures are reserved for "unifying" talks on topics of interest to the whole range of ETAPS attendees. The aim of cramming all this activity into a single one-week meeting is to create a strong magnet for academic and industrial researchers working on topics within its scope, giving them the opportunity to learn about research in related areas, and thereby to foster new and existing links between work in areas that were formerly addressed in separate meetings.

ETAPS 2003 was organized by Warsaw University, Institute of Informatics, in cooperation with the Foundation for Information Technology Development, as well as:

- European Association for Theoretical Computer Science (EATCS);
- European Association for Programming Languages and Systems (EAPLS);
- European Association of Software Science and Technology (EASST);
- ACM SIGACT, SIGSOFT and SIGPLAN.

The organizing team comprised:

Mikołaj Bojańczyk, Jacek Chrząszcz, Piotr Chrząstowski-Wachtel, Grzegorz Grudziński, Kazimierz Grygiel, Piotr Hoffman, Janusz Jabłonowski, Mirosław Kowaluk, Marcin Kubica (publicity), Sławomir Leszczyński (www), Wojciech Moczydłowski, Damian Niwiński (satellite events), Aleksy Schubert, Hanna Sokołowska, Piotr Stańczyk, Krzysztof Szafran, Marcin Szczuka, Łukasz Sznuk, Andrzej Tarlecki (co-chair), Jerzy Tiuryn, Jerzy Tyszkiewicz (book exhibition), Paweł Urzyczyn (co-chair), Daria Walukiewicz-Chrząszcz, Artur Zawłocki.

ETAPS 2003 received support from:[1]

- Warsaw University
- European Commission, High-Level Scientific Conferences and Information Society Technologies
- US Navy Office of Naval Research International Field Office,
- European Office of Aerospace Research and Development, US Air Force
- Microsoft Research

Overall planning for ETAPS conferences is the responsibility of its Steering Committee, whose current membership is:

Egidio Astesiano (Genoa), Pierpaolo Degano (Pisa), Hartmut Ehrig (Berlin), José Fiadeiro (Leicester), Marie-Claude Gaudel (Paris), Evelyn Duesterwald (IBM), Hubert Garavel (Grenoble), Andy Gordon (Microsoft Research, Cambridge), Roberto Gorrieri (Bologna), Susanne Graf (Grenoble), Görel Hedin (Lund), Nigel Horspool (Victoria), Kurt Jensen (Aarhus), Paul Klint (Amsterdam), Tiziana Margaria (Dortmund), Ugo Montanari (Pisa), Mogens Nielsen (Aarhus), Hanne Riis Nielson (Copenhagen), Fernando Orejas (Barcelona), Mauro Pezzè (Milano), Andreas Podelski (Saarbrücken), Don Sannella (Edinburgh), David Schmidt (Kansas), Bernhard Steffen (Dortmund), Andrzej Tarlecki (Warsaw), Igor Walukiewicz (Bordeaux), Herbert Weber (Berlin).

I would like to express my sincere gratitude to all of these people and organizations, the program committee chairs and PC members of the ETAPS conferences, the organizers of the satellite events, the speakers themselves, and Springer-Verlag for agreeing to publish the ETAPS proceedings. The final votes of thanks must go, however, to Andrzej Tarlecki and Paweł Urzyczyn. They accepted the risk of organizing what is the first edition of ETAPS in Eastern Europe, at a time of economic uncertainty, but with great courage and determination. They deserve our greatest applause.

Leicester, January 2003

José Luiz Fiadeiro
ETAPS Steering Committee Chair

[1] The contents of this volume do not necessarily reflect the positions or the policies of these organizations and no official endorsement should be inferred.

Preface

The conference on Fundamental Approaches to Software Engineering (FASE) aims at presenting novel results and discussing new trends in both theories for supporting software engineering and experiences of application of theories for improving software engineering practice.

This year, the conference focused on large-scale information and communication infrastructures evolving in time and functionalities and with stringent quality constraints. The international community answered enthusiastically with a record of 89 high-quality submissions in many key areas, covering both traditional and emerging technologies: software components and architectures, mobile computing, aspect programming, object-oriented programming, distributed and web computing, model integration, software measurements, analysis, and testing. Contributions were submitted from 27 different countries, covering Europe (Italy, Germany, United Kingdom, France, Belgium, Spain, Portugal, Poland, Austria, Cyprus, Finland, Ireland, Luxemburg, Norway, Russia, and Sweden), America (United States, Canada, Brazil, Argentina, and Chile), Asia (Korea, Japan, India, China, Jordan), and Australia.

Each submission was reviewed by three independent reviewers. The quality was high, the competition was strong, and the Program Committee worked hard to select only the top-quality submissions, allowing time for the other authors to assess the results that we hope to see presented soon to our community. The resulting program included 26 papers and was introduced by a keynote address from Michal Young.

We would like to thank all authors who submitted their work for presentation at FASE. Without their excellent contributions, the work of the Program Committee would have been tedious and we would not have been able to prepare such a great program. We express our gratitude to the Organizing and Steering Committees, who gave us excellent support. A special thanks to the members of the Program Committee and the many reviewers who supported a smooth and exciting reviewing process.

Milan, January 2003 Mauro Pezzé

Organization

Program Chair

Mauro Pezzè	Università degli Studi di Milano Bicocca (Italy)

Program Committee

Luciano Baresi	Politecnico di Milano (Italy)
Andrea Corradini	Università degli Studi di Pisa (Italy)
Hartmut Ehrig	Technical University of Berlin (Germany)
José Fiadeiro	University of Leicester (UK)
Istvan Forgàs	Balthazar (Hungary)
Marie-Claude Gaudel	Université de Paris-Sud (France)
Heinrich Hussmann	Dresden University of Technology (Germany)
Mehdi Jazayeri	Technical University of Vienna (Austria)
Lee Osterweil	University of Massachusetts (USA)
Gianna Reggio	Università degli Studi di Genova (Italy)
Richard Taylor	University of California, Irvine (USA)
Andy Schürr	Darmstadt University of Technology (Germany)
Roel Wieringa	University of Twente (The Netherlands)

Table of Contents

Keynote

Software Components

Mobile Computing

Aspect and Object-Oriented Programming

Distributed and Web Applications

Software Measurements

Formal Verification

Analysis and Testing

Model Integrations and Extensions

Symbiosis of Static Analysis and Program Testing

Michal Young

University of Oregon, Dept. of Computer Science
michal@cs.uoregon.edu

1 Introduction

The fundamental fact about verifying properties of software, by any means, is that almost anything worth knowing is undecidable in principle. The limitations of software testing, on the one hand, and static analysis on the other, are just different manifestations of this one basic fact. Because both approaches to verification are ultimately doomed, neither is likely to supplant the other in the foreseeable future. On the other hand, each can complement the other, and some of the most promising avenues of research are in combinations and hybrid techniques.

2 Limits of Dynamic Testing

The basic theory underlying testing is almost entirely negative. One can of course resort to randomized testing if the objective is to measure rather than to improve the product, but the number of test cases required to obtain high levels of confidence is astronomical [5]. Moreover, statistical inferences are valid only if one has a valid statistical model of use, which is rare. More often testing is systematic, not random, and aimed at finding faults rather than estimating the prevalence of failures.

Systematic testing exercises divides the set of possible executions into a finite number of classes, and inspects samples from each class on the hope that classes are sufficiently homogeneous to raise the likelihood of fault detection. Thus systematic testing, whether based on program structure or specification structure or something else, is based on a *model* of software and a *hypothesis* that faults are somehow localized by the model. This hypothesis is not verifiable except individually and after the fact, by observing whether some test obligation induced from the model was in fact effective in selecting fault-revealing test cases.

3 Limits of Static Analysis

The unsatisfying foundations of testing make exhaustive, static analyses seem more attractive. Isn't it better to achieve some kind of result that is sound, even if we must accept some spurious error warnings, or restrict the class of programs

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 1–5, 2003.

to be analyzed, or check properties that are simpler than those we really want to analyze? Indeed, some properties can be incorporated into the syntax or static semantics of a programming language and checked every time we compile a program, like Java's requirement to initialize each variable before use and to explicitly declare the set of exceptions that can be thrown or propagated by a program unit.

Exhaustive static analyses are necessarily based on abstract models of the software to be checked. The static checks that have been built into Java are based on simple models derived from program syntax. The "initialize before use" rule is based on a simple, local control flow model, and crucially it sidesteps the fundamental limitation of undecidability by not distinguishing between executable and unexecutable program paths. It is acceptable for a Java compiler to reject a program with a syntactic path on which the first use of a variable appears before it is assigned a variable, even if that path can never be taken, because the restriction is easy to understand and the "fault" is easy to repair. The rule regarding declaration of exceptions is likewise based on a simple, easily understandable call-graph model. Because the rules can be understood in relation to these models, we accept them as being rather strict but precise, rather than viewing them as producing spurious error reports.

More sophisticated program checks – for example, checking synchronization structure to verify absence of race conditions – place much heavier demands on extraction of an appropriate model. If the model is too simple, the analysis is apt to be much too pessimistic, and unlike the Java rules mentioned above there may be no reasonable design rule to prevent spurious error reports. If the model is sufficiently expressive to avoid spurious error reports, an exhaustive analysis is likely to be unacceptably expensive. The remaining option is to use an expressive model, but limit analysis effort by other means, such as using a non-exhaustive (and unsound) analysis, in which case the "static" analysis becomes a kind of symbolic testing.

Sometimes the model is produced by a human, and in that case again the limits on analysis are not too onerous if failures are reported in the form of counter-examples whose cause is easily diagnosed. Crafting a model with appropriate abstractions to efficiently verify properties of interest is a challenging design task, but arguably at least the effort is recouped in establishing a clearer understanding of the software system [16]. A remaining problem is that one cannot be certain of the correspondence between the model and the actual software system it represents. Verifying their correspondence has, roughly speaking, the same difficulty as extracting models directly from software, and raises essentially the same issues.

Despite the surge of interest in static analysis of models derived directly (with varying degrees of automation) from actual software source code [8,14,11,2], the fundamental limitation imposed by undecidability ensures that static analysis will not supplant dynamic testing anytime soon. In some application domains, it may be possible to impose enough restrictions on programs that testing is relegated to a minor role. One can imagine severe restrictions on programming

style in software destined for medical devices, for example. In the larger world of software development, the trends are in the wrong direction. Dynamically reconfigurable systems, programs that migrate across a network, end-user programmable applications, and aspect-oriented programming (to pick a few) all widen the gap between the kinds of programs that developers write, and the kinds of programs that are amenable to strong static analysis.

4 Symbiotic Interactions

Testing has limitations, and static analysis has limitations, but it does not follow immediately that some combination or hybrid of testing and analysis should be better than either one independently. One could imagine that, while neither is perfect, one dominates the other. But this does not turn out to be the case, and there are many current examples of symbiotic interactions between static analysis and testing as well as additional opportunities for fruitful combinations.

One interesting class of combination is an individual technique that combines aspects of both, exploring selected scenarios like testing but using a symbolic representation of program state more like a pure static analysis technique. Symbolic testing techniques have the same ultimate limitation as conventional testing, but using symbolic representations they can more effectively search for particular classes of program fault. Howden developed a symbolic testing technique more than 25 years ago [13], but program analysis technology and computing power was perhaps not ready for it then. Lately the technique has been revived and elaborated, notably in Pincus' Prefix [4] and Engler's Metal [7].

Symbolic testing, like conventional testing, is not sound: If we fail to find a violation of some property of interest, that does not constitute a proof of the property. Exhaustive static analyses, on the other hand, are typically designed to be sound, at least with respect to the abstract model on which they operate. As we noted above, this leaves the problem of discrepancies between the model and the underlying program. Relegating the problem of model conformance to dynamic testing, as in communication protocol conformance testing, is an attractive option. Separating analysis of the model from dynamic testing of model conformance makes each simpler. It makes diagnosing problems in the model much easier, and provides much more flexibility (e.g., in the use of pointers and other dynamic structures) than insisting that the programmer use only idioms that can be automatically (and soundly) abstracted to a model.

Dynamic testing often divides the execution space of a program into putative equivalence classes based on the results of some form of static analysis of program text, e.g., data flow testing based on def-use associations produced by a data flow analysis. To date, most such testing techniques have been based on conservative analyses. For example, points-to analysis typically overestimates the set of pointers that can point to the same object, resulting in spurious def-use associations; this in turn can lead to test obligations that cannot be met. While conservative analysis can be justified when used directly to find faults, it

makes more sense to use a precise but unsafe analysis to provide guidance for testing [12].

One could take this a step further: It is easy to imagine static program analyses that are unsound, but which produce as a by-product a set of unverified assumptions to be checked by dynamic testing. For example, one might choose to perform a static analysis that ignores exceptions or some aliasing relations, except for those that have been observed in dynamic tests. Or, one could produce static analysis results in which "maybe" results are distinguished from definite faults, as Chechik and Ding have done with model checking [6], and use "maybe" results as guidance to testing.

Even a simple, intraprocedural control flow graph mode model is overly conservative in that it includes program paths that cannot actually be executed, so systematic testing typically ends with a residue of unmet coverage obligations. This residue itself can be considered as a kind of model or hypothesis about behavior in actual use. If we release a product with some unmet coverage obligations, we are hypothesizing that they are either impossible to execute (an artifact of an overly conservative model) or at least so rarely executed as to be insignificant. This hypothesis can be tested in actual use, through program instrumentation [15,3].

Increased computational power and clever algorithms benefit dynamic testing and program monitoring, just as they have benefited static analysis and particularly finite-state verification techniques. One of the techniques that would have seemed implausible a decade ago is dynamically gathering "specifications" (more precisely, hypothetical properties) during program execution, as in Ernst's Daikon [9] and the "specification mining" technique of Ammons, Bodik, and Larus [1]. These can be applied directly to testing [10]. An interesting possibility is to make these highly likely but unproven properties available to static analysis techniques, again giving up soundness (when necessary) to achieve more precise analysis. It is also possible that, once identified, some of them might be statically verified.

Many years ago, a battle raged between the partisans of program verification and of dynamic testing. Thankfully, that battle is long over, and static and dynamic analysis techniques (as well as other aspects of formal methods, such as precise specification) are almost universally regarded as complementary. Models are the common currrency of static analysis and dynamic testing. By accepting the inevitability of imperfect models, we open many opportunites for synergistic combinations.

References

1. Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16. ACM Press, 2002.
2. T. Ball and S. Rajamani. Checking temporal properties of software with boolean programs. In *Proceedings of the Workshop on Advances in Verification*, 2000.

3. Jim Bowring, Alessandro Orso, and Mary Jean Harrold. Monitoring deployed software using software tomography. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 2–9. ACM Press, 2002.
4. William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software – Practice and Experience*, 30(7):775–802, 2000.
5. Ricky W. Butler and George B. Finelli. The infeasibility of quantifying the reliability of life-critical real-time software. *IEEE Transactions on Software Engineering*, 19(1):3–12, 1993.
6. M. Chechik and W. Ding. Lightweight reasoning about program correctness. Technical Report CSRG Technical Report 396, University of Toronto, 2000.
7. Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in Metal. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 51–60. ACM Press, 2002.
8. James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zhenga. Bandera: Extracting finite-state models from Java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
9. Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.
10. Michael Harder. Improving test suites via generated specifications. Master's thesis, M.I.T., Dept. of EECS, May 2002.
11. Gerard J. Holzmann and Margaret H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Transactions on Software Engineering*, 28(4):364–377, 2002.
12. J. Horgan and S. London. Data flow coverage and the C language. In *Proceedings of the Fourth Symposium on Testing, Analysis, and Verification*, pages 87–97, Victoria, Oct 1991. ACM Press.
13. William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *IEEE Transactions on Software Engineering*, SE–3(4):266–278, July 1977.
14. D. Park, U. Stern, and D. Dill. Java model checking. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification*, Limerick, Ireland, June 2000.
15. Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *International Conference on Software Engineering*, pages 277–284, 1999.
16. Wei Jen Yeh and Michal Young. Redesigning tasking structures of Ada programs for analysis: A case study. *Software Testing, Verification, and Reliability*, 4:223–253, December 1994.

An Ontology for Software Component Matching

Claus Pahl

Dublin City University, School of Computer Applications
Dublin 9, Ireland
Claus.Pahl@dcu.ie

Abstract. The Web is likely to be a central platform for software development in the future. We investigate how Semantic Web technologies, in particular ontologies, can be utilised to support software component development in a Web environment. We use description logics, which underlie Semantic Web ontology languages such as DAML+OIL, to develop an ontology for matching requested and provided components. A link between modal logic and description logics will prove invaluable for the provision of reasoning support for component and service behaviour.

1 Introduction

Component-based Software Engineering (CBSE) increases the reliability and maintainability of software through reuse [1,2]. Providing reusable software components and plug-and-play style software deployment is the central objective. CBSE originates from the area of object-oriented software development, but tries to overcome some of the problems associated with object-orientedness [1]. Components are software artefacts that can be individually developed and tested. Constructing loosely coupled software systems by composing components is a form of software development that is ideally suited for development in distributed environments such as the Web. Distributed component-based software development is based on component selection from repositories and integration.

Reasoning about component descriptions and component matching is a critical activity [3]. Ontologies are knowledge representation frameworks defining concepts and properties of a domain and providing the vocabulary and facilities to reason about these. Two ontologies are important for the component context:

- *Application domain ontologies* describe the domain of the software application under development.
- *Software development ontologies* describe the software development entities and processes.

The need to create a shared understanding for an application domain is long recognised. Client, user and developer of a software system need to agree on concepts for the domain and their properties. Domain modelling is a widely used requirements engineering technique. However, with the emergence of distributed software development and CBSE also the need to create a shared understanding of software entities and development processes arises. We will present here a

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 6–21, 2003.

software development ontology providing the crucial matching support for CBSE that is a substantial step ahead compared to the reasoning capabilities of current matching approaches such as DAML-S for Web Services [4,5].

Component matching techniques are crucial in Web-based component development. Providing component technology for the Web requires to adapt to Web standards. Since semantics are particularly important, ontology languages and theories of the Semantic Web initiative [6] need to be adopted. Formality in the Semantic Web framework facilitates machine understanding and automated reasoning. The ontology language DAML+OIL is equivalent to a very expressive description logic [7,8]. This fruitful connection provides well-defined semantics and reasoning systems. Description logics provide a range of class constructors to describe concepts. Decidability and complexity issues – important for the tractability of the technique – have been studied intensively.

Description logic is particularly interesting for the software engineering context due to a correspondence between description logics and modal logic [8,9]. The correspondence between description logics and dynamic logic (a modal logic of programs) is based on a similarity between quantified constructors (expressing quantified relations between concepts) and modal constructors (expressing safety and liveness properties of programs). We aim to enable the specification of transition systems in description logic. This enables us to reason about service and component behaviour. We present a novel approach to Web component matching by encoding transitional reasoning about safety and liveness properties – essentially from dynamic logic which is a modal program logic [10] – into a description logic and ontology framework.

We focus on the description of components and their services and their relation to the Semantic Web in Sect. 2. Reasoning about matching is the content of Sect. 3. We end with a discussion of related work and some conclusions.

2 Service and Component Description

2.1 The Component Model

Different component models are suggested in the literature [1,2,11,12]. Here is an outline of the key elements of our component model:

- *Explicit export and import interfaces.* In particular explicit and formal import interfaces make components more context independent. Only the properties of required services and components are specified.
- *Semantic description of services.* In addition to syntactical information such as service signatures, the abstract specification of service behaviour is a necessity for reusable software components.
- *Interaction patterns.* An interaction pattern describes the protocol of service activations that a user of a component has to follow in order to use the component in a meaningful way.

An example that illustrates our component model – see Fig. 1 – consists of a service requestor and a service provider component. The interface allows users to

Component DocInterface	*Component* DocStorageServer
import services	*import services*
create(id:ID)	...
retrieve(id:ID):Doc	*export services*
update(id:ID,upd:Doc)	crtDoc(id:ID)
preCond valid(upd)	rtrDoc(id:ID):Doc
postCond retrieve(id)=upd	updDoc(id:ID,upd:Doc)
export services	*preCond* wellFormed(upd)
openDoc(id:ID)	*postCond* rtrDoc(id)=upd∧wellFormed(upd)
saveDoc(id:ID, doc:Doc)	delDoc(id:ID)
import interaction pattern	*export interaction pattern*
create;!(retrieve+update)	crtDoc;!(rtrDoc+updDoc);delDoc

Fig. 1. Document processing example

open and save documents; it requires services from a suitable server component to create, retrieve, and update documents. The server provides a range of services. An empty document can be created using `crtDoc`. The request service `rtrDoc` retrieves a document, but does not change the state of the server component, whereas the update service `updDoc` updates a stored document without returning a value. Documents can also be deleted. A requirements specification of a service user for an `update` service is given. If documents are XML-documents, these can be well-formed (correct tag nesting) or valid (well-formed and conform to a document type definition DTD). We have specified an import interaction pattern for client `DocInterface` and for provider `DocStorageServer` an export pattern. The import pattern means that the `create` service is expected to be executed first, followed by a repeated invocation of either `retrieve` or `update`.

2.2 An Ontology for Component Description

The starting point in defining an ontology is to decide what the basic ontology elements – concepts and role – represent. Our key idea is that the ontology formalises a software system and its specification, see Fig. 2. Concepts represent component system properties. Importantly, systems are dynamic, i.e. the descriptions of properties are inherently based on an underlying notion of state and state change. Roles represent two different kinds of relations. *Transitional roles* represent accessibility relations, i.e. they represent processes resulting in state changes. *Descriptional roles* represent properties in a given state.

We develop a description logic to define the component matching ontology. A description logic consists of three types of entities. Individuals can be thought of as constants, concepts as unary predicates, and roles as binary predicates. Concepts are the central entities. They can represent anything from concrete objects of the real world to abstract ideas.

De nition 1. Concepts *are collections or classes of objects with the same properties. Concepts are interpreted by sets of objects.* **Individuals** *are named objects.* **Concept descriptions** *are formed according to the following rules: A is*

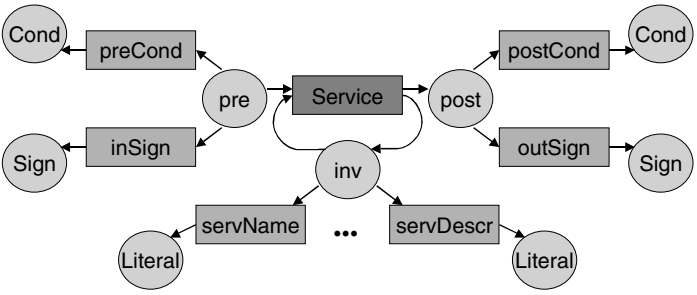


Fig. 2. Software development ontology

an atomic concept, and if C and D are concepts, then so are $\top$, $\bot$, $\neg C$, and $C \sqcap D$. Combinators such as $C \sqcup D$ or $C \rightarrow D$ are defined as usual. **Roles** *are relations between concepts.*

Roles allow us to associate properties to concepts. Two basic forms of role applications are important for our context. These will be made available in form of concept descriptions.

Definition 2. *Value restriction and existential quantification extend the set of concept descriptions*[1]*. A* **value restriction** $\forall R \triangleright C$ *restricts the value of role R to elements that satisfy concept C. An* **existential quantification** $\exists R \triangleright C$ *requires the existence of a role value. Quantified roles can be composed. Since $\forall R_2 \triangleright C$ is a concept description, the expression $\forall R_1 \triangleright \forall R_2 \triangleright C$ is also a concept description.*

Example 1. An example for the value restriction is $\forall$`preCond.wellFormed`: all conditions are well-formed. An existential quantification is $\exists$`preCond.wellFormed`: there is at least one condition `preCond` that is well-formed.

The constructor $\forall R \triangleright C$ is interpreted as either an accessibility relation R to a new state C for transitional roles such as `update`, or as a property R satisfying a constraint C for descriptional roles such as `postCond`.

Example 2. For a transitional role `update` and a descriptional role `postCond`, the expression $\forall$`update`$\triangleright\forall$`postCond`$\triangleright$`equal(retrieve(id),doc)` means that by executing service `update` a state can be reached that is described by the post-condition `equal(retrieve(id),doc)` – an element of a condition domain.

2.3 Interpretation of Concepts and Roles

We interpret concepts and roles in Kripke transition systems [10]. Kripke transition systems are semantical structures used to interpret modal logics that are

[1] In description logic terminology, this language is called $\mathcal{ALC}$, which is an extension of the basic attributive language $\mathcal{AL}$.

also suitable to interpret description logics. Concepts are interpreted as states. Transitional roles are interpreted as accessibility relations.

Definition 3. *A* **Kripke transition system** $M = (\mathcal{S}, \mathcal{L}, \mathcal{T}, I)$ *consists of a set of states* $\mathcal{S}$*, a set of role labels* $\mathcal{L}$*, a transition relation* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$*, and an interpretation* I*. We write* $R_{\mathcal{T}} \subseteq \mathcal{S} \times \mathcal{S}$ *for a transition relation for role* R*.*

The set $\mathcal{S}$ interprets the state domains *pre*, *post*, and *inv* – see Fig. 2. Later we extend the set $\mathcal{S}$ of states by several auxiliary domains such as *Cond*, *Sign*, or *Literal* that represent description domains for service and component properties.

Definition 4. *For a given Kripke transition system* M *with interpretation* I*, we define the model-based* **semantics of concept descriptions***:*

$$
\begin{aligned}
\top &= \mathcal{S} \\
\bot &= \emptyset \\
(\neg A)^I &= \mathcal{S} \backslash A^I \\
(C \sqcap D)^I &= C^I \cap D^I \\
(\forall R \triangleright C)^I &= \{a \in S | \forall b \triangleright (a, b) \in R^I \rightarrow b \in C^I\} \\
(\exists R \triangleright C)^I &= \{a \in S | \exists b \triangleright (a, b) \in R^I \wedge b \in C^I\}
\end{aligned}
$$

A notion of undefinedness or divergence exists in form of bottom $\bot$. Some predefined roles, e.g. the identity role *id* interpreted as $\{(x, x) | x \in \mathcal{S}\}$, shall be introduced. The predefined descriptional roles are defined as follows: $preCond^I \subseteq pre^I \times Cond^I$, $inSign^I \subseteq pre^I \times Sign^I$, $postCond^I \subseteq post^I \times Cond^I$, $outSign^I \subseteq post^I \times Sign^I$, $servName^I \subseteq inv^I \times Literal^I$, $servDescr^I \subseteq inv^I \times Literal^I$.

The semantics of description logics is usually given by interpretation in models. However, it can also be defined by translation into first-order logic [8]. Concepts C can be thought of as unary predicates $C(x)$. Roles R can be thought of as binary relations $R(x, y)$. Then, $\forall R \triangleright C$ corresponds to $\forall x \triangleright R(y, x) \rightarrow C(x)$.

2.4 Role Constructs and Component Behaviour

Expressive role constructs are essential for our application. *Transitional roles* $R_{\mathcal{T}}$ represent component services: $(R_{\mathcal{T}})^I \subseteq \mathcal{S} \times \mathcal{S}$. They are interpreted as accessibility relations on states. *Descriptional roles* $R_{\mathcal{D}}$ are used to describe properties of services dependant on the state: $(R_{\mathcal{D}})^I \subseteq \mathcal{S} \times \mathcal{D}$ for some auxiliary domain $\mathcal{D}$. These are interpreted as relations between states and property domains. In our case, the set of descriptive roles is fixed (*preCond*, *postCond*, *inSign*, *outSign*, etc.), whereas the transitive roles are application-specific services.

An ontology for component matching requires an extension of basic description logics by composite roles that can represent interaction patterns [8].

Definition 5. *The following* **role constructors** *shall be introduced:*

- $R ; S$ **sequential composition** *with* $(R ; S)^I = \{(a, c) \in S^I \times S^I | \exists b \triangleright (a, b) \in R^I \wedge (b, c) \in S^I\}$*; often we use* $\circ$ *instead of* $;$ *to emphasise functional composition*

- $!R$ **iteration** *with* $!R^I = \bigcup_{i \geq 1}(R^I)^i$*, i.e. the transitive closure of* R^I
- $R + S$ **non-deterministic choice** *with* $(R + S)^I = R^I \cup S^I$

Expressions constructed from role names and role constructors are **composite roles**. $P(R_1, \ldots, R_n)$ *is an abstraction refering to a composite role* P *based on the atomic roles* $R_1, \ldots, R_n$.

Example 3. The value restriction $\forall$ `create;!(retrieve+update)` $\triangleright$`postState` is based on the composite role `create;!(retrieve+update)`.

Definition 6. *A* **role chain** $R_1 \circ \ldots \circ R_n$ *is a sequential composition of functional roles (roles that are interpreted by functions).*

Axioms in our description logic allow us to reason about service behaviour. Questions concerning the consistency and role composition with respect to pre- and postconditions can be addressed.

Proposition 1. *Selected properties of quantified descriptions: (i)* $\forall R \triangleright \forall S \triangleright C \Leftrightarrow R; S \triangleright C$*, (ii)* $\forall R \triangleright C \sqcap D \Leftrightarrow \forall R \triangleright C \sqcap \forall R \triangleright D$*, (iii)* $\forall R \sqcup S \triangleright C \Leftrightarrow \forall R \triangleright C \sqcup \forall S \triangleright C$.

Proof. Follows from proofs from dynamic logic axioms such as $[p][q] \Leftrightarrow [p;q]$ for (i) – see [10] Theorem 3. □

A special form of a role constructor is the existential predicate restriction. This will be needed in conjunction with concrete domains – see Sect. 2.6.

Definition 7. *The role expression* $\exists(u_1, \ldots, u_n) \triangleright P$ *is an* **existential predicate restriction**, *if* P *is an* n*-ary predicate of a concrete domain – concepts can only be unary – and* $u_1, \ldots, u_n$ *are role chains. Analogously, we define the* **universal predicate restriction** $\forall(u_1, \ldots, u_n) \triangleright P$.

$\exists(x, y) \triangleright equal$ expresses that there are role fillers for the two roles x and y that are equal. The expression $\forall(x, y) \triangleright equal$ requires all role fillers to be equal.

2.5 Names and Parameterisation

Individuals are introduced in form of assertions. For instance `Doc(D)` says that individual `D` is a document `Doc`. `length(D,100)` says that the length of `D` is 100.

Definition 8. Individual *x with $C(x)$ is interpreted by $x^I \in \mathcal{S}$ with $x^I \in C^I$.*

It is also possible to introduce individuals on the level of concepts and roles.

Definition 9. *The* **set constructor**, *written* $\{a_1, \ldots, a_n\}$ *introduces the individual names* $a_1, \ldots, a_n$. *The* **role filler** $R : a$ *is defined by* $(R : a)^I = \{b \in \mathcal{S} | (b, a^I) \in R^I\}$, *i.e. the set of objects that have* a *as a filler for* R.

This means that $R : a$ and $\exists R \triangleright \{a\}$ are equivalent.

The essential difference between classical description logic and our variant here is that we need names to occur in role and concept descriptions. A description logic expression $\forall \texttt{create} \triangleright \texttt{valid}$ usually means that `valid` is a concept, or predicate, that can be applied to some individual object; it can be thought of as $\forall \texttt{create(x)} \triangleright \texttt{valid(x)}$ for an individual `x`. If roles are services, then `x` should not represent a concrete individual, but rather a name or a variable. For instance the document creation service `create` has a parameter `id`.

Our objective is to introduce names into the description language. The role filler construct provides the central idea for our definition of names.

Definition 10. *We denote a* **name** n *by a role* $n[Name]$*, defined by* $(n[Name])^I = \{(n^I \vartriangleleft n^I)\}$*. A* **parameterised role** *is a transitional role* R *applied to a name* $n[Name]$*, i.e.* $R \circ n[Name]$*.*

In first-order dynamic logic, names are identifiers interpreted in a non-abstract state. These names would have associated values, i.e. a state is a mapping (binding of current values). However, since we are going to define names as roles this explicit state mapping is not necessary.

Proposition 2. *The name definition* $n[Name]$ *is derived from the role filler and the identity role definition:* $(n[Name])^I(n^I) = (id : n)^I$*.*

Proof. $(n[Name])^I(n^I) = \{(n^I \vartriangleleft n^I)\}(n^I) = \{n^I\} = \{n^I | (n^I \vartriangleleft n^I) \in id^I\} = \{b | (b \vartriangleleft n^I) \in id^I\} = (id : n)^I$. □

The idea of presenting names as roles is borrowed from category theory[2].

We can now express a parameterised role $\forall \texttt{create} \circ \texttt{id}[Name] \triangleright \texttt{post}$ defined by $\{x | \forall y \triangleright (x \vartriangleleft y) \in (\texttt{create} \circ \texttt{id}[Name])^I \rightarrow y \in \texttt{post}^I\}$ which is equal to $\{\texttt{id}^I | y \in \texttt{post}^I\}$, where y is a *postState* element that could be further described by roles such as $y = \forall \texttt{postCond} \triangleright \texttt{post} \sqcap \forall \texttt{outSign} \triangleright \texttt{out}$. The expression $\texttt{create} \circ \texttt{id}[Name]$ is a role chain, assuming that `create` is a functional role: $(\texttt{create} \circ \texttt{id}[Name])^I = \{(a \vartriangleleft c) | (a \vartriangleleft b) \in \texttt{id}[Name]^I \wedge (b \vartriangleleft c) \in \texttt{create}^I\} = \{(\texttt{id}^I \vartriangleleft p) | (\texttt{id}^I \vartriangleleft \texttt{id}^I) \in \texttt{id}[Name]^I \wedge (\texttt{id}^I \vartriangleleft p) \in \texttt{create}^I\} = \{(\texttt{id}^I \vartriangleleft p)\}$.

Example 4. With names and role composition the following *parameterised role chain* can now be expressed:

$$\forall\ \texttt{update} \circ (\texttt{id}[Name] \vartriangleleft \texttt{doc}[Name]); \texttt{postCond} \triangleright \texttt{equal(retrieve(id),doc)}$$

Note, that we often drop the $[Name]$ annotation if it is clear from the context that a name is under consideration.

[2] A point in category theory [13] resembles our name definition. A point in the category of finite sets is an arrow from a singleton set **1** to another object. This arrow can be seen as a mapping giving a name to a target value.

2.6 Concrete Domains and Property Types

Concrete domains and predefined predicates for these domains have been proposed to add more concrete elements to descriptions [8]. A classical example is to introduce a numerical domain with predicates such as $\leq$, $\geq$ or equality. These predicates can be used in the same way as concepts – which can also be thought of as unary predicates. An example is $\texttt{Doc} \sqcap \exists \texttt{length} \triangleright \geq 100$ where the last element is a predicate $\{n | n \geq 100\}$. `length` is a functional role, i.e. an attribute which maps to a concrete domain. Binary predicates such as equality can be used in conjunction with predicate restriction role constructors, e.g. $\exists(\texttt{x} \texttt{y}) \triangleright \texttt{equal}$.

Definition 11. Concrete domains *are interpreted by algebraic structures with a base set and predicates interpreted as n-ary relations on that base set.*

Concrete domains are important in our context since they allow us to represent application domain-specific knowledge. These domains will be refered to by type names. Concrete domains are needed for all application-oriented types used in a component specification.

Example 5. The update service deals with two types of entities: documents and identifiers. The *document domain* $\texttt{Doc} \equiv \exists \texttt{hasStatus} . \texttt{valid} \sqcup \texttt{wellFormed}$ and $\texttt{valid} \sqsubseteq \texttt{wellFormed}$ describes documents. Two predicates `valid` and `wellFormed` exist, which are in a subsumption or subclass relation. For the *identifier domain* `ID` only a binary predicate `equal` shall be assumed.

2.7 Contracts and Interaction Patterns

Axioms are introduced into description logics to reason about concept and role descriptions.

Definition 12. *Subconcept* $C1 \sqsubseteq C2$*, concept equality* $C1 \equiv C2$*, subrole* $R1 \sqsubseteq R2$*, role equality* $R1 \equiv R2$*, and individual equality* $\{x\} \equiv \{y\}$ *are* **axioms**. *The semantics of these axioms is defined based on set inclusion of interpretations for* $\sqsubseteq$ *and equality for* $\equiv$.

All forms of axioms are reducible to subsumption, i.e. subconcept or subrole [8]. Description logics often introduce an equivalence of concepts often as a definition in a macro style – the left-hand side is a new symbol, e.g. $\texttt{Status} \equiv \texttt{valid} \sqcup \texttt{wellFormed}$.

Contractual service descriptions form the basis of the matching of component services represented by atomic roles. The specification of `update` using axioms in description logic in Fig. 3 illustrates this. **Interaction patterns** can be specified using composite roles, e.g. $\forall \texttt{create} \circ \texttt{id}; !(\texttt{retrieve} \circ \texttt{id} + \texttt{update} \circ (\texttt{id} \texttt{doc})) \triangleright \texttt{post}$. It describes the interaction protocol that a component can engage in. There is one import interaction pattern and one export interaction pattern for each component.

The logic allows us to specify both safety and liveness properties of services.

$$
\begin{array}{lcl}
\texttt{pre} & \equiv & \forall \texttt{preCond.valid(doc)} \\
 & & \sqcap\ \forall \texttt{inSign.(id : ID, doc : Doc)} \\
 & & \sqcap\ \forall \texttt{update} \circ \texttt{(id, doc).post} \\
\texttt{post} & \equiv & \forall \texttt{postCond.equal(retrieve} \circ \texttt{id, doc)} \\
 & & \sqcap\ \forall \texttt{outSign.()} \\
\texttt{inv} & \equiv & \forall \texttt{servName.\{"update"\}} \\
 & & \sqcap\ \forall \texttt{servDescr.\{"updates document"\}} \\
 & & \sqcap\ \forall \texttt{update} \circ \texttt{(id, doc).inv}
\end{array}
$$

Fig. 3. Contractual description of service `update`

```
<daml:Class>
   <daml:Restriction>
       <daml:onProperty rdf:resource="#update(id,doc)"/>
       <daml:toClass>
           <daml:unionOf rdf:parseType="daml:collection">
               <daml:Restriction>
                  <daml:onProperty rdf:resource="#postCond"/>
                  <daml:hasClass rdf:resource="#equal(retrieve(id),doc)"/>
               </daml:Restriction>
               <daml:Restriction>
                  <daml:onProperty rdf:resource="#outSign"/>
                  <daml:hasClass rdf:resource="#()"/>
               </daml:Restriction>
           </daml:unionOf>
       </daml:toClass>
   </daml:Restriction>
</damL:Class>
```

Fig. 4. DAML+OIL specification

Example 6. We can express that eventually after executing *create*, a document will be deleted: $(\forall \texttt{preCond} \triangleright \texttt{true}) \sqcap (\forall \texttt{create} \triangleright \exists \texttt{delete} \triangleright \forall \texttt{postCond} \triangleright \texttt{true})$[3].

2.8 DAML+OIL and the Semantic Web

The Semantic Web initiative bases the formulation of ontologies on two Web technologies for content description: XML and RDF/RDF Schema. RDF Schema is an ontology language providing classes and properties, range and domain notions, and a sub/superclass relationship. Web ontologies can be defined in DAML+OIL – an ontology language whose primitives are based on

[3] This corresponds to a dynamic logic formula $\texttt{true} \rightarrow [\texttt{create(id)}]\langle \texttt{delete(id)} \rangle\, \texttt{true}$ combining safety ($[\ldots]\phi$) and liveness ($\langle \ldots \rangle \psi$) properties.

XML and RDF/RDF Schema, which provides a much richer set of description primitives. DAML+OIL can be defined in terms of description logics [14]. However, DAML+OIL uses a different terminology; corresponding notions are class/concept or property/role. We present the DAML+OIL specification of formula $\forall$`update` $\circ$ (`id`$\triangleleft$`doc`)$\triangleright$($\forall$`postCond`$\triangleright$`equal(retrieve(id),doc)` $\sqcap$ $\forall$`outSign`$\triangleright$`()`) in Fig. 4.

3 Inference and Matching

The two problems that we are concerned with are component description and component matching. Key constructs of description logics to support this are equivalence and subsumption. In this section, we look at component matching based on contracts and how it relates to subsumption reasoning.

3.1 Subsumption

Subsumption is defined by subset inclusions for concepts and roles.

Definition 13. *A* **subsumption** $C_1 \sqsubseteq C_2$ *between two concepts* C_1 *and* C_2 *is defined through set inclusion for the interpretations* $C_1^I \subseteq C_2^I$*. A* **subsumption** $R_1 \sqsubseteq R_2$ *between two roles* R_1 *and* R_2 *holds, if* $R_1^I \subseteq R_2^I$*.*

Subsumption is not implication. Structural subsumption (subclass) is weaker than logical subsumption (implication), see [8].

Proposition 3. *The following axioms hold for concepts* C_1 *and* C_2*: (i)* $C_1 \sqcap C_2 \sqsubseteq C_1$*, (ii)* $C_1 \wedge C_2 \rightarrow C_1$*, (iii)* $C_2 \rightarrow C_1$ *implies* $C_2 \sqsubseteq C_1$*.*

Proof. (i) $C_1 \sqcap C_2 \sqsubseteq C_1$ is true since $C_1^I \cap C_2^I \subseteq C_1^I$. *(ii)* $C_1 \wedge C_2 \rightarrow C_1$ is true since $(C_1 \wedge C_2)^I \subseteq C_1^I$. *(iii)* $C_2 \rightarrow C_1$ implies $C_2^I \subseteq C_1^I$ since structural subsumption is weaker than logical subsumption. □

We can use subsumption to reason about matching of two service descriptions (transitional roles).

3.2 Matching of Services

Subsumption is the central reasoning concept in description logics. We will integrate service reasoning and component matching with this concept.

A service is functionally specified through pre- and postconditions. Matching of services is defined in terms of implications on pre- and postconditions and signature matching based on the widely accepted design-by-contract approach[4]. The CONS inference rule, found in dynamic logic [10], describes the refinement of services. Based on the hypotheses $\square \rightarrow \square' \triangleleft \square \rightarrow [p]\ \square \triangleleft$ and $\square' \rightarrow \square$ we can conclude $\square' \rightarrow [p]\square'$. A matching definition for services, i.e. transitional roles, shall be derived from the CONS rule.

[4] We ignore other descriptions such as invariants and possible improvements of our refinement notion through subsignatures here.

Definition 14. *A provided service P* **refines** *a requested service R, or service P* **matches** *R, if*

$$\frac{\forall inSign \triangleright in_R \sqcap \forall R \triangleright \forall outSign \triangleright out_R}{\forall inSign \triangleright in_P \sqcap \forall P \triangleright \forall outSign \triangleright out_P} \langle\ in_P \equiv in_R \wedge out_P \equiv out_R$$

(signatures are compatible: types of corresponding parameters are the same) and

$$\frac{\forall preCond \triangleright pre_R \sqcap \forall R \triangleright \forall postCond \triangleright post_R}{\forall preCond \triangleright pre_P \sqcap \forall P \triangleright \forall postCond \triangleright post_P} \langle\ pre_R \sqsubseteq pre_P \wedge post_P \sqsubseteq post_R$$

(requested service precondition is weakened and postcondition strengthened).

Matching of service descriptions is refinement. This is a contravariant inference rule that captures service matching based on formal behaviour specification.

Example 7. The service `updDoc` of the document server matches the requirements of `update` – a service that might be called in methods provided by the interface. Signatures are compatible. `updDoc` has a weaker, less restricted precondition (`valid(doc)` implies `wellFormed(doc)`) and a stronger postcondition (the conjunction `retrieve(id)=doc` ∧ `wellFormed(doc)` implies `retrieve(id)=doc`). This means that the provided service satisfies the requirements.

Proposition 4. *The matching rule for services defined in Definition 14 is sound.*

Proof. (i) Assume that $\forall preCond \triangleright pre_R$ and $pre_R \sqsubseteq pre_P$. Then $preCond^I = \{(a^\varsigma b) | b \in pre_R^I\}$ and $pre_R^I \subseteq pre_P^I$ implies $preCond^I = \{(a^\varsigma b) | b \in pre_P^I\}$ for $\forall preCond \triangleright pre_P$. (ii) Assume that $\forall R; postCond \triangleright post_R$ and $post_P \sqsubseteq post_R$. The former implies that $(R; postCond)^I = \{(a^\varsigma c) | (\exists b \triangleright (a^\varsigma b) \in R^I \wedge (b^\varsigma c) \in postCond^I) \wedge c \in post_R^I\}$ and $post_P^I \subseteq post_R^I$ implies that $(P; postCond)^I = \{(a^\varsigma c) | (\exists b' \triangleright (a^\varsigma b') \in P^I \wedge (b'^\varsigma c) \in postCond^I) \wedge c \in post_P^I\}$ for $\forall P; postCond \triangleright post_P$ for a role P that is syntactically compatible with R. □

Matching implies subsumption, but is not the same. Refinement (matching of services) is a sufficient criterion for subsumption.

Proposition 5. *If service P refines (or matches) R, then $P \sqsubseteq R$.*

Proof. If P refines R, i.e. $pre_R^I \subseteq pre_P^I$ and $post_P^I \subseteq post_R^I$, then for each $(a^\varsigma b) \in P^I$ there is an $(a^\varsigma b) \in R^I$. Therefore, $P^I \subseteq R^I$, and consequently $P \sqsubseteq R$. □

If the conditions are application domain-specific, e.g. predicates such as `valid(doc)`, then an underlying domain-specific theory provided by an application domain ontology can be integrated via concrete domains.

3.3 Matching of Interaction Patterns

Together with service matching based on contractual descriptions, interaction pattern matching is the basis of component matching.

A notion of consistency of composite roles relates to the underlying service specifications based on e.g. pre- and postconditions.

Definition 15. *A composite role* $P(R_1 \leftthreetimes \triangleright\triangleright\triangleright \leftthreetimes R_n)$ *is* **consistent**, *if the last state is reachable. A concept description* $\forall P(R_1 \leftthreetimes \triangleright\triangleright\triangleright \leftthreetimes R_n) \triangleright\!\!\triangleleft C$ *with transitional role* P *is* **reachable** *if* $\{(a \leftthreetimes b) \in P^I | \exists b \triangleright\!\!\triangleleft b \in C^I\}$ *is not empty.*

Proposition 6. *A composite role* P *is* **consistent** *if the following (sufficient) conditions are satisfied:*

(i) for each sequence $R;S$ *in* P : $\forall postCond \triangleright post_R \sqsubseteq \forall preCond \triangleright pre_S$
(ii) for each iteration $!R$ *in* P : $\forall postCond \triangleright post_R \sqsubseteq \forall preCond \triangleright pre_R$
(iii) for each choice $R+S$ *in* P : $\forall preCond \triangleright pre_R \sqcap \forall preCond \triangleright pre_S$ *and* $\forall postCond \triangleright post_R \sqcap \forall postCond \triangleright post_S$

Proof. By definition $(R;S)^I = \{(a \leftthreetimes c) \in (R;S)^I | \exists b \triangleright\!\!\triangleleft (a \leftthreetimes b) \in R^I \wedge (b \leftthreetimes c) \in S^I\}$. Then $\forall postCond \triangleright post_R^I \subseteq \forall preCond \triangleright pre_S^I$ implies that each $b \in \forall postCond \triangleright post_R^I$ is also in $\forall preCond \triangleright pre_S^I$, i.e. $b \in \forall postCond \triangleright post_R^I \Rightarrow b \in \forall preCond \triangleright pre_S^I$. Similarly for $!R$ since $!R = R; \triangleright\triangleright\triangleright; R$. For each $R+S$ both pre- and postconditions need to be enabled to guarantee successful execution. □

Definition 16. *A component* **interaction pattern** *is a consistent composite role* $P(R_1 \leftthreetimes \triangleright\triangleright\triangleright \leftthreetimes R_n)$ *constructed from transitional role names and the connectors* ; , | , *and* $+$[5]. *Interaction patterns are interpreted by* **pattern transition graphs** *for composite transitional roles, i.e. the graphs that represent all possible pattern executions.*

Both client and provider components participate in interaction processes based on the services described in their import and export interfaces. The client will show a certain import interaction pattern, i.e. a certain ordering of requests to execute provider services. The provider on the other hand will impose a constraint on the ordering of the execution of services that are provided.

The specification of interaction patterns describes the ordering of observable activities of the component process. Process calculi suggest simulations and bisimulations as constructs to address the equivalence of interaction patterns. We will use a notion of simulation between processes to define interaction pattern matching between requestor and provider.

Definition 17. *A provider interaction pattern* $P(S_1 \leftthreetimes \triangleright\triangleright\triangleright \leftthreetimes S_k)$ **simulates** *a requested interaction pattern* $R(T_1 \leftthreetimes \triangleright\triangleright\triangleright \leftthreetimes T_l)$, *or pattern* P **matches** R, *if there exists a homomorphism* θ *from the transition graph of* R *to the transition graph of* P, *i.e. if for each* $R_g \xrightarrow{T_i} R_h$ *there is a* $P_k \xrightarrow{S_j} P_l$ *such that* $R_g = \theta(P_k)$, $R_h = \theta(P_l)$, *and* S_j *refines* T_i.

[5] We often drop service parameters in patterns since only the ordering is relevant.

Matching of interaction patterns is simulation. The form of this definition originates from the simulation definition of the □-calculus, see e.g. [15]. Note, that simulation subsumes service matching. The provider needs to be able to simulate the request, i.e. needs to meet the expected interaction pattern of the requestor.

The definition implies that the association between S_i and T_j is not fixed, i.e. any S_i such that S_i refines T_j for a requested service T_j is suitable. For a given T_j, in principle several different provider services S_i can provide the actual service execution during the process execution.

Example 8. The provider requires `crtDoc;!(rtrDoc+updDoc);delDoc` and the requestor expects `create;!(retrieve+update)` as the ordering. Assuming that the service pairs `crtDoc`/`create`, `rtrDoc`/`retrieve`, and `updDoc`/`update` match based on their descriptions, we can see that the provider matches (i.e. simulates) the required server interaction pattern. The `delDoc` service is not requested.

As for service matching we expect interaction pattern matching not to be the same as subsumption. Subsumption on roles is input/output-oriented, whereas the simulation needs to consider internal states of the composite role execution. For each request in a pattern, there needs to be a corresponding provided service. However, matching is again a sufficient condition for subsumption.

Proposition 7. *If the component interaction pattern* $P(S_1 \triangleleft \triangleright\triangleright\triangleright \triangleleft S_k)$ *simulates the interaction pattern* $R(T_1 \triangleleft \triangleright\triangleright\triangleright \triangleleft T_l)$*, then* $R \sqsubseteq P$*.*

Proof. If $P(S_1 \triangleleft \triangleright\triangleright\triangleright \triangleleft S_k)$ simulates $R(T_1 \triangleleft \triangleright\triangleright\triangleright \triangleleft T_l)$, then for each $(a \triangleleft b) \in R^I$ there is a pair $(a \triangleleft b) \in P^I$. Therefore, $R^I \subseteq P^I$, and consequently $R \sqsubseteq P$ follow. □

Note, that the provider might support more transitions, i.e. subsumes the requestor, whereas for service matching, the requestor subsumes the provider (the provider needs to be more specific).

3.4 Complexity and Decidability

The tractability of reasoning about descriptions is a central issue for description logic. The richness of our description logic has some negative implications for the complexity of reasoning. However, some aspects help to reduce the complexity. We can restrict roles to functional roles. Another beneficial factor is that for composite roles negation is not required. We do not investigate this aspect in depth [8] – only one issue shall be addressed.

A crucial problem is the decidability of the specification if concrete domains are added. Admissible domains guarantee decidability.

Definition 18. *A domain D is called* **admissible** *if the set of predicate names is closed under negation, i.e. for any n-ary predicate P there is a predicate Q such that* $Q^D = (S^D)^n \backslash P^D$*, there is a name* $\top_D$ *for* S^D*, and the satisfiability problem is decidable; i.e. there exists an assignment of elements of* S^D *to variables such that the conjunction* $\wedge_{i=1}^{k} P_i(x_1^{(i)} \triangleleft \triangleright\triangleright\triangleright \triangleleft x_{n_i}^{(i)})$ *of predicates* P_i *becomes true in D.*

Proposition 8. *We can show that our chosen concrete domains (documents and identifiers) – see Example 5 – are admissible.*

Proof. In [8], it is shown that the domain $\mathcal{N}$ with the set of nonnegative integers and the predicates $<^{\iota} \leq^{\iota} >^{\iota} \geq$ is admissible. We can map documents and identifiers to nonnegative numbers and lexicographical ordering predicates to the binary predicates. Consequently, the domains are admissible. □

4 Related Work

The formula ∀`update ∘ (id,doc)`⊳∀`postCond`⊳`equal(retrieve(id),doc)` in description logic corresponds to [`update(id,doc)`][`postCond`] `retrieve(id)=doc` in dynamic logic. Schild [9] points out that some description logics are notational variants of multi-modal logics. This correspondence allows us to integrate modal axioms and inference rules about programs or processes into description logics. We have expanded Schild's results by addressing the problem of representing names in the notation and by defining a matching inference framework.

Some effort has already been made to exploit Semantic Web and ontology technology for the software domain [4,5,16]. All of these approaches have so far focused on the restricted component-aspects of Web services. [16] addresses the configuration of Web services; [5] presents solutions in the DAML-S context, which is the closest project to our work.

DAML-S [4] is a DAML+OIL ontology for describing properties and capabilities of Web services. DAML-S represents services as classes (concepts). Knowledge about a service is divided into two parts. A service profile is a class that describes what a service requires and what it provides, i.e. external properties. A service model is a class that describes workflows and possible execution paths of a service, i.e. properties that concern the implementation. DAML-S relies on DAML+OIL subsumption reasoning to match requested and provided services. DAML-S [4] provides to some extend for Web services what we aim at for Web components. However, the form of reasoning and ontology support that we have provided here is not possible in DAML-S, since services are modelled as concepts and not rules in the DAML-S ontology. Only considering services as roles makes modal reasoning about process behaviour possible.

5 Conclusions

Component development lends itself to development by distributed teams in a distributed environment. Reusable components from repositories can be bound into new software developments. The Web is an ideal infrastructure to support this form of development. We have explored Semantic Web technologies, in particular description logics that underlie Web ontology languages, for the context of component development. Ontologies can support application domain modelling, but we want to emphasise the importance of formalising central development activities such as component matching in form of ontologies.

Adding semantics to the Web is the central goal of the Semantic Web activity. Our overall objective has been to provide advanced reasoning power for a semantic Web component framework. We have presented description logic focussing on semantical information of components. The behaviour of components is essentially characterised by the component's interaction processes with its environment and by the properties of the individual services requested or provided in these processes. The reasoning capabilities that we have obtained and represented in form of a matching ontology go beyond current ontologies for service matching. Even though description logics have been developed to address knowledge representation problems in general, the connection to modal logics has allowed us to obtain a rich framework for representing and reasoning about components. Description logic is central for two reasons. Firstly, it is a framework focusing strongly on the tractability of reasoning, and, secondly, it is crucial for the integration of component technology into the Web environment.

Some questions have remained unanswered. Decidability and complexity results from description logic need to be looked at in more detail. We plan to adapt exiting proof techniques and to use description logic systems such as FaCT.

References

1. C. Szyperski. *Component Software: Beyond Object-Oriented Programming – 2nd Ed.* Addison-Wesley, 2002.
2. G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems.* Cambridge University Press, 2000.
3. A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Trans. on Software Eng. and Meth.*, 6(4):333–369, 1997.
4. DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
5. J. Peer. Bringing Together Semantic Web and Web Services. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
6. W3C Semantic Web Activity. Semantic Web Activity Statement, 2002. `http://www.w3.org/sw`.
7. I. Horrocks, D. McGuiness, and C. Welty. Digital Libraries and Web-based Information Systems. In F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors, *The Description Logic Handbook.* Cambridge University Press, 2003.
8. F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003. (to appear).
9. K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence.* 1991.
10. Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.
11. C. Pahl. Components, Contracts and Connectors for the Unified Modelling Language. In *Proc. Symposium Formal Methods Europe 2001, Berlin, Germany.* Springer-Verlag, LNCS-Series, 2001.

12. M. Casey. *Towards a Web Component Framework: an Investigation into the Suitability of Web Service Technologies for Web-based Components.* M.Sc. Dissertation. Dublin City University, 2002.
13. F.W. Lawvere and S. Schanuel. *Conceptual Mathematics.* Cambridge University Press, 1998.
14. DAML Initiative. *DAML+OIL Ontology Markup.* `http://www.daml.org,2001`.
15. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
16. A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Semantic Configuration Web Services in the CAWICOMS Project. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.

A Description Language for Composable Components

Ioana Şora[1], Pierre Verbaeten[2], and Yolande Berbers[2]

[1] Universitatea Politehnica Timisoara, Department of Computer Science
Bd. V.Parvan 2, 1900 Timisoara, Romania
ioana@cs.utt.ro
[2] Katholieke Universiteit Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Leuven, Belgium
{pierre.verbaeten,yolande.berbers}@cs.kuleuven.ac.be

Abstract. In this paper[1] we present *CCDL*, our description language for *composable* components. We have introduced hierarchically composable components as means to achieve finetuned customization of component based systems. A composable component is defined by a fixed contractual specification of its external view and a set of structural constraints for its internal configuration. The internal configuration of a composable component is not fixed, but will be composed according to different requirements and has to comply with the structural constraints. This permits a high degree of unanticipated variability. Our approach is architectural style specific and addresses multiflow architectures. The goal of CCDL is to describe contractual specifications and structural constraints of composable components, as guidelines for their composition. CCDL descriptions can be used by automatic composition tools that implement requirements driven composition strategies.

1 Introduction

The growing complexity of modern software systems has lead to an emphasis on compositional approaches for system development. The advantages of a component-based approach to software engineering are twofold. Firstly, it can reduce production time through discipline and re-use, leading to more manageable systems. Secondly, such assembled systems can be easily updated by changing one or more components. These changes make it possible to respond to unanticipated future requirements or to a larger and better offer on the component market.

An important research topic is to be able to predict the properties of a component assembly, based on the properties of its components. In the case of *software composition*, predictable assembly means to find a set of components and to determine the collaborations between them so that it forms a software

[1] This research has been partially carried out in order of Alcatel Bell with financial support of the Flemish institute for the advancement of scientific-technological research in the industry (IWT SCAN # 010319)

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 22–36, 2003.

system that complies with a given set of requirements. However, composing a whole system only from its requirements is not feasible, additional constraints and guidelines are needed.

Automatic component composition can be used as a mechanism to achieve (dynamic) self-customizable systems that are able to adapt themselves to changing user requirements or to their evolving environment. The general issues of software composition apply as well in the case of automatic component composition. Also, there is a need to support unanticipated customization: solutions should not be limited to the use of a set of known in advance components or configurations. Solutions must be open to discover and integrate new components and configurations, in response to new types of requests or to improve existing solutions when new components become available. The problem that arises here is to balance between the support for unanticipated customizations and the need for constraints that guarantee a correct composition of a system with required properties. Such constraints are essential especially in the case of non-computable properties, that means when the properties of the assembly cannot be calculated from individual components properties. The fact that in case of automatic component composition the composition decision is a machine decision requires additional rigor and thoroughness. Automatic software composition has to be built on a systematic *compositional model* that must comprise a component description scheme and formalism, and a coordinated, well defined requirements driven composition strategy.

We developed a composable components model, together with CCDL as its description formalism, in the context of a compositional model for self-customizable systems. The internal configuration of a *composable* component, while not fixed, must comply to a fixed set of structural constraints, part of the component description. These structural constraints are only guidelines for future composition of the internal structure and not a full configuration description. The structural constraints are flexible enough to allow unanticipated compositions. The actual internal structure of a composable component is dynamically determined according to current requirements. This component description model establishes what information is needed to be known about the components in order to make composition decisions while CCDL defines a formalism that can be processed by automatic composition tools. Compositional decisions are made in knowledge of the architectural style, but ignoring some details of the underlying component technology, as long as this complies with the architectural assumptions. Working at the architectural level has the advantage that it abstracts domain-specific issues and permits implementation of generic composition strategies. We are developing a compositional model, targeted at *multi-flow architectures*. This establishes a framework for finding a component composition with desired properties, based on properties of individual components. We have built a *Composer* tool to automatize the requirements driven composition of systems.

The remainder of this paper is organized as follows: Section 2 presents the overall picture of our automatic component composition approach. Section 3

introduces the basic concepts of our architectural component model. We describe the composable component approach and CCDL in Sect. 4. Section 5 presents deployment scenarios for CCDL descriptions. In Sect. 6 we discuss our approach in the context of related work. The last section presents the concluding remarks.

2 Background

The automatic composition problem, as we address it, is the following: *given a set of requirements describing the properties of a desired system, and a component repository that contains descriptions of available components, the composition process has to find a set of components and their configuration to realize the desired system.* All components can be composable, that means that determining their internal structure is also a recursive composition problem.

The compositional decision making system (the *Composer*) operates on an architectural model [OGT+99] of the system. This architectural model is a structure description of the composed system. The Composer finds the structure of the target system starting from the imposed requirements. The Composer is architecture-style specific, the composition decisions implemented by the Composer do not contain application-specific code. The Composer determines and maintains the structure description of the composed system, while a *Builder* uses this structure description to build or maintain the system. The Builder depends upon (or is part of) the underlying component technology and framework. This integrated approach for self-customizable systems is depicted in Fig. 1.

The Composer operates with requirements stated as expressions that contain component properties. In the case when self-customization is used as start-up time configuration according to user requirements, an optional *Translator* front-end (i.e., in form of a Wizard) could be deployed to generate the requirements in form of properties expressions from a more domain-specific form.

The Composer has access to a repository containing CCDL descriptions of available components. The target of the composition is also a composable component and it has its structural constraints described in CCDL. The structural constraints have the role of guidelines for the composition, as they will be discussed further in Sect. 4.2

As we describe in Sect. 3.2, our model comprises simple and composable components. The composition will result through stepwise refinements: after a composition process has determined that it wants a certain component type in place, and this is a composable one, a new composition search may be launched for composing the internal structure of it, in order to finetune its properties.

We have used the Composer to achieve self customizable network protocol stacks, as presented in [SMBV02] and [SVB02].

3 The Architectural Component Model

The compositional model proposed by us assumes that the system architecture belongs to a certain architectural style. Some details of the component model

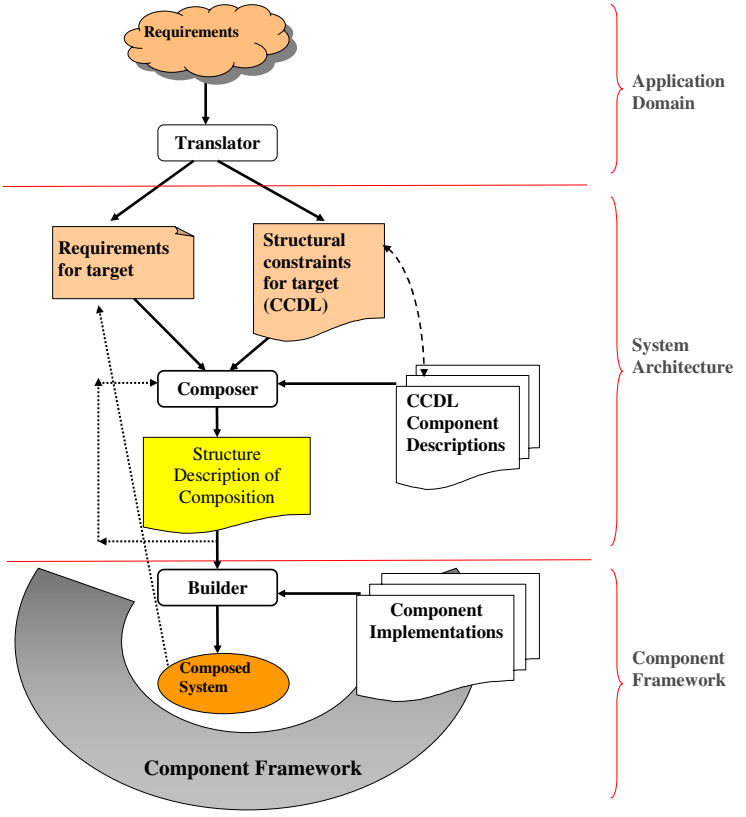


Fig. 1. Integrated approach for self-customizable systems

(like the programming interfaces) are not presented here, since they are not used in the composing phase, only later in the building phase of the system (as mentioned before in Sect. 2).

3.1 Basic Concepts

We present briefly the basic component concepts that we use and that are consistent, in the main, with the software component bibliography [BBB+00,Szy97]. We emphasize here our personal interpretations and particularities.

Software component: is an implementation of some functionality, available under the condition of a certain contract, independently deployable and subject to composition. A component in our approach is also an architectural abstraction.

Component contract: specifies the services provided by the component and the obligation of clients and environment needed by the component to provide

these services. In our approach, contracts are expressed through sets of required-provided properties.

Component property: "Something that is known and detectable about the component" [HMSW02]. In our approach, a property is expressed through a name (a label) from a *vocabulary* set. These names are treated in a semantic-unaware way [SMBV02]. In our more recent approach, values can be also associated with properties.

Port: "a logical point of interaction between the component and its environment" [AG97]. There are input ports, through which the component receives data, and output ports, through which the component generates data.

Flow: the data-flow relation among pairs of ports. A flow has parts where it is internal to a component (from an input to an output port of that component) and parts where it is between two components (a connection).

Multi-flow architecture: it is a variation of the pipes-and-filters [Gar01] architecture. An informal example is presented in Fig. 2. The particularity of this architectural style is that dataflow relations are defined first (the "flows" in our terminology) and components must fit on them. For every component the internal flows must be known so that they can be integrated in the flow architecture. In the example in Fig. 2, the system S has two flows on which it contains the subcomponents (Fig. 2a). The system S can be realized as different compositions of components on these flows, two possibilities are depicted in Fig. 2b.

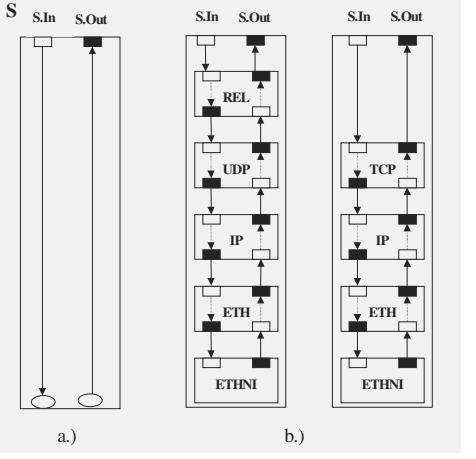


Fig. 2. Multi-flow architecture example. System S is defined by two flows and can be realized as different component compositions on these

3.2 Hierarchical Relationships between Components

Our model comprises simple and composed components. A simple component is the basic unit of composition, has one input port and one output port. A composed component appears as a grouping mechanism, may have several input and output ports. The whole system may be seen as a composed component, as S being both the system and a composed component in the example in Fig. 2. The internal structure of a composed component is aligned on a number of flows that connect its input ports with some of its output ports. For the internal structure of a composed component the same style of multi-flow architecture applies.

Composed components are "first class" components, they have their own properties and contractual interfaces and fixed internal flows. The composed component as a whole is always defined by its own set of provided properties, which expresses the higher-abstraction-level features gained through the composition of the subcomponents. The vocabulary used to describe the own provides of a composed component is distinct from the vocabulary deployed for describing the provides of its subcomponents. This abstraction definition must be done by the designer of the composed component. The properties of the subcomponent are causally linked to the properties of the composed component, but often they cannot be computed or deducted from them, as it is the case for example with the functional properties. Many properties of an assembly are emergent properties, they are related to the overall behavior of the assembly and depend on the collaboration of several components and can be seen as expressed at a higher abstraction level.

The internal structure of a composed component is mostly not fixed, these components are *composable* in the limits of certain structural constraints, as will be presented in Sect. 4.2.

3.3 Contracts Expressed through Properties

Contracts are expressed through required-provided properties. Provided properties are associated with the component as a whole. Requirements are associated with the ports. A contract for a component is respected if all its required properties have found a match. Requirements associated with an input port $C_x \triangleright In_y$ are addressed to components which have output ports connected to the flow ingoing $C_x \triangleright In_y$. Requirements associated with an output port $C_x \triangleright Out_y$ are addressing components which have inputs connected to the flow exiting $C_x \triangleright Out_y$.

In the case of composed components, provided properties can also be associated with ports, reflecting the internal structure of the component.

We assume that in the underlying component model, every input port may be connected to every output port. The meaningful compositions are determined by the criteria of correct composition, based on matching required with provided properties. A properties match is primarily defined by the match of the properties names. Properties values, if present, are used as parameters for further component configuration. By default, it is sufficient that requirements are met by some components that are present in the flow connected to that

port, these requirements are able to *propagate*. The mechanism of propagation of requirements and the basic composition strategy are presented in [SMBV02]. One can specify *immediate* requirements, which are not propagated, these apply only to the next component on that flow. *Negative* requirements specify that a property should not be present on the referred flow. *Pair* requirements refer to pairs that must be always matched in the same relative order.

As an example, in Fig. 3 is presented a simple assembly of fully matched components. The example presents a data flow part of a sender-receiver system, where encryption and compression of the transmitted data must occur and also the data transmission time must be calculated. The example system in the figure comprises seven components, and the assembly fulfills the system requirements and all component requirements are fully matched. The $TripTimeCalculator$ component calculates the time delay on a given flow. It requires that timestamps are attached to the data on its incoming flow (has the requirement *timestamp* at its input port $TripTimeCalculator.In$. This is a propagateable requirement, it can be provided by a component at any place in the incoming flow of $TripTimeCalculator$, as it is the case with component $Timestamper$ that provides the property *timestamp*. The $Encrypter$ component has the requirement for *decryption* on its outgoing flow, declared as a *pair* requirement. Also the $Compresser$ has a pair requirement for *decompression*. Since requirements declared as pairs must be matched in the same order as they were posed, the $Compresser - Decompresser$ sequence may either contain the $Encryptor - Decryptor$ sequence or be contained by it. A sequence like $Encryptor - Compresser - Decryptor - Decompresser$ is not permitted due to the requirements being declared as pair.

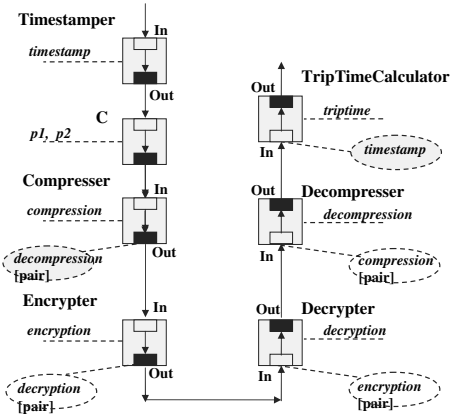


Fig. 3. Contracts expressed through required-provided *properties*

4 CCDL – The Description Language for Composable Components

We have developed *CCDL*, a description language that allows the specification of *composable* components. Current interface definition languages do not provide sufficient information about the described components for automatic component composition. Architectural description languages do not support the undefined variable internal structure of composable components. As presented in [MT00], a condition for a language to be an ADL is to be able to represent components, connectors and architectural configurations. Architectural configurations describe an architectural structure (topology) of components and connectors. The point is, in our case, the internal structure of composable components needs to be *not* represented, its configuration must remain open. In CCDL, only "structural constraints" have to be specified, leaving open the possibilities of composition. The structural constraints in our component descriptions are just flexible guidelines for future configuration compositions and not a full architecture configuration description.

We have defined *CCDL*, our component description language for composable components, as a XML Schema. The XML Schema standard is a meta-language suitable for developing new notations. This choice for XML [XML00] simplifies the implementation and later the use of the description language due to the large availability of tools for creating, editing and manipulating XML documents. We use the Apache Xerces XML parser in our implementation. Editing a CCDL description can be done with aid of syntax-directed editing tools for XML.

The CCDL description of a component comprises two main parts: the external view (contractual specification through information about global provided properties and information on the ports) and the internal view, if it is a composed component. The internal view may specify either structural constraints, if the component is composable, or a complete structural description, if the component has an already fixed internal configuration. The description of the external and internal view will be detailed in the next sections.

4.1 External View Description

The component externals contain the set of provided properties and information about all the ports. Sets of properties are used to describe the component contracts – for requirements and provides. As mentioned before, properties are in the essence names, and they are treated in a semantic-unaware way by the Composer. Properties can be further specified by values that are configuration parameters. Properties may come with a list of subproperties (introduced by the WITH tag after a property definition). The list of subproperties represents finetuning options. Subproperties are often used to finetune a requirement addressed to a composable component. I.e., for a requirement $p1\ WITH\ p11, p12$, a match is a component C that exposes the provided property $p1$, and the internal structure of C must be further composed so that it will provide the finetuning properties $p11$ and $p12$. Subproperties can also be present in the definition of a

composed component if its structure is already fixed. A composable component has usually no subproperties in its definition.

4.2 Internal View Description

The composable components do not have a fixed internal structure. In this approach lies a powerful part of the customization capability: the full internal configuration of the component will be composed as a result of external requirements allowing fine-tuning of properties [SJBV02,SVB02].

As mentioned before (in Sect. 3.2), composed components are first class entities, that have their own interfaces with ports, provided and required properties. They have also well defined internal flows. These are the fixed elements of a composed component. The internal structure is not fixed, but still some *structural constraints* must exist in order to always enforce compliance with the global component description.

The structural constraints of a composable component comprise: basic structural constraints, structural context-dependent requirements for components and inter-flow dependencies.

The *basic structural constraints* describe the minimal properties that must be assembled on particular flows for the declared provides of the composed component to emerge. These constraints virtually define a "skeleton" of the composed component. This "skeleton" is not a rigid structure, it fixes only the flows and establishes order relationships between properties that must be present on these flows. Figure 4 presents as an example the basic structural constraints for a component S, that represents a customizable network protocol stack. The basic structural constraints of S specify that it has two flows, corresponding to the downgoing (sending) and upgoing (receiving) path of packets through the stack. On these flows arbitrary protocol layer components can be added. The structural constraints specify that at the bottom of the stack the property $NetwInt$ must be provided.

The basic structural constraints must be specified by the developer of the composed component.

The *structural context-dependent requirements* express requirements related to other components when deployed here as subcomponents. If a certain component X can act as a subcomponent in the context of a composed component C, and the basic structural constraints of C are not sufficient to fully specify the correct deployment of X in the context of C, then appropriate specifications have to be added to the structural context-dependent requirements of C. These structural context-dependent requirements in C will be added by the developer of the subcomponent X. Syntactically, they are expressed also in terms of properties contained on flows and order relationships between these properties. The presence of these contained properties or order relationships as context-dependent requirements does not impose a mandatory skeleton as for the basic structural constraints. They specify the terms under which a certain subcomponent may be deployed here, but only if it considered necessary by external requirements.

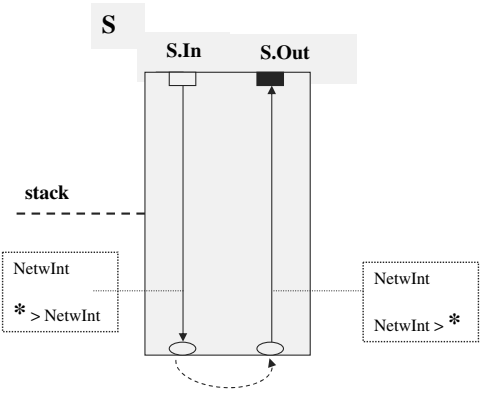


Fig. 4. Basic structural constraints example for composable component S

The *inter-flow dependencies* specify relationships between the flows. These relationships between flows can express continuation (a flow might be a logical continuation of another) or connections between flows. In the example in Fig. 4, there is a logical continuation relationship between the two flows.

The full CCDL description of S, the composable stack component, is given in Fig. 5.

A concrete stack will be built after determining its structure according to specific requirements. For example, if the current requirements are:
stack $WITH$ *rel*, *transp*, two possible component assemblies that match these requirements are these depicted in Fig. 2b.

5 Repositories

The deployment of components is supported by information from a *component repository* and an *implementation repository*. The component repository contains CCDL descriptions of components. The implementation repository contains implementation descriptions for existing component description. Not every component description must have a known implementation, the composable components usually do not have known implementations. Different ways of implementation descriptions are possible: by having specified the implementation classes or referring to the structure of a composed component. The implementation description may add also the implementation characteristics – a set of properties particular only for the implementation. This decoupling between component descriptions and implementation descriptions facilitates an easy deployment as well for using components as for introducing new component types.

The decision to use an existing component is made based on the component description. Later, an implementation must be found for that component, using the implementation repository. While the choice of a component made on hand of its properties handles the functional features of the composed system, non-functional requirements are handled by the choice of a right implementation,

```
<?xml version="1.0" encoding="UTF-8"?>
<!--CCDL for composable Stack component-->
<component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="E:\comp.xsd" name="Stack">
    <componentExternals>
        <provides>
            <property name="stack"/>
        </provides>
        <port name="in" type="in" entrance="true"/>
        <port name="out" type="out"  entrance="true"/>
    </componentExternals>
    <componentInternals>
        <structuralConstraints>
            <basicStructuralConstraints>
                <start name="start_up">
                    <out name="out"/>
                </start>
                <end name="end_down">
                    <in name="in"/>
                </end>
                <flow name="downgoing" from="in" to="end_down.in"/>
                <flow name="upgoing" from="start_up.out" to="out"/>
                <containedProperty name="nw_interface" flowlocation="downgoing"/>
                <containedProperty name="nw_interface" flowlocation="upgoing"/>
                <orderRelation below="nw_interface" above="any" flowlocation="downgoing"/>
                <orderRelation below="any" above="nw_interface" flowlocation="upgoing"/>
            </basicStructuralConstraints>
            <interflowDependencies>
                <continuation from="downgoing" to="upgoing"/>
            </interflowDependencies>
            <contextDependencies/>
        </structuralConstraints>
    </componentInternals>
</component>
```

Fig. 5. CCDL description

based on the implementation characteristics. If there is no known implementation, or if the implementation is not according to all the requirements, the component will be composed according to the requirements and in the limits of the structural constraints specified in the component description.

New components may be easily introduced to the system, following a scenario like depicted in Fig. 6, that presents the actions to make a new component CX known to the system. First, the new component has to be described, using for its properties terms from the established vocabulary, or extending the vocabulary when introducing new properties. The vocabulary of properties can be extended with new properties definitions. However, the extension of the vocabulary as well as deploying properties in component description should be subject to certification. Secondly, a CCDL description of the new component must be added. It may appear the need to define special interactions for the component CX when used in certain contexts. This leads to an update of the structural requirements of the components where it might be deployed. It is the task of the designer of the component CX to provide a list of updates for the use of CX in the

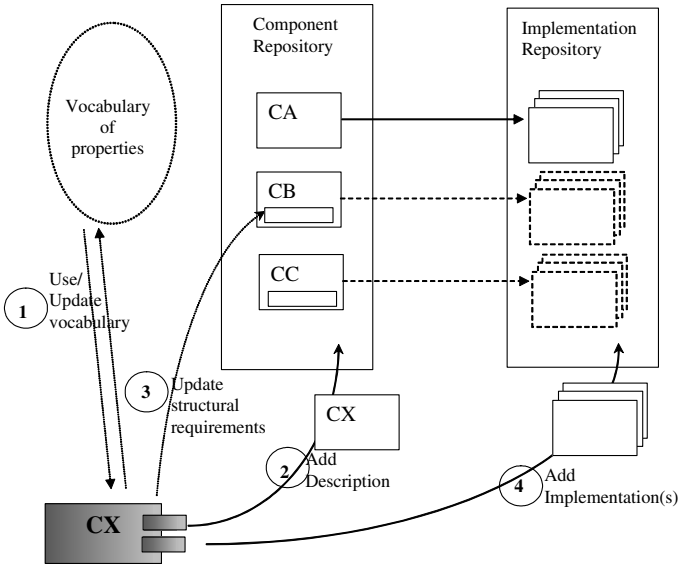


Fig. 6. Deployment of repository tools. Example of introducing a new component CX to the system

context of different compositions. For example, when using CX in the composition of CB, special restrictions might appear as to where CX should be placed in CB's internal structure. This is the case when CX provides new properties, which were unknown at the moment of CB's development. Ordering relations on the flows between these new properties and the other properties that might be involved must be specified. Final step is to provide implementation(s) for it, either in form of implementation classes or a complete description of the internal structure of a composed component. Composed components do not necessarily have implementations specified.

6 Discussion and Related Work

Our insight is that a composition model should address the architectural level, to be usable across different applications that share that architectural style. The approach is to build a system by assuming a certain defined architectural style. This approach of component composition being treated in the context of architecture is largely accepted in the research community [Ham02,IT02b, Wil01,IS01,BG97,KI00], as it makes the problem manageable and eliminates the problems of architectural mismatch [GAO95]. In this context, we present a model for composable components in multiflow architectures, together with CCDL, its description language.

CCDL is neither an interface description language nor an architectural description language, but presents some concepts related with both of them. Issues

of composition of architectural components have been addressed within ADL's (see [MT00] for an comprehensive overview). In these cases, deciding a good component combination is done statically and relies completely on the application programmer. Even within ADL's that support dynamic architectures, the dynamism is a "programmed" one. Here the goal of CCDL is different than that of ADL's. The role of ADL's is to model and describe software architectures, with their explicit configuration. The information contained in an architectural description can be used in tools to analyze properties of the architectural structure. On the other hand, CCDL does not fully describe a composed system, neither a composed component. It states only guidelines for future composition of that system or component, in form of structural constraints. The role of CCDL is to describe minimal requirements for the system configuration, leaving the configuration itself open. Tools take CCDL descriptions and generate the concrete structural configuration according to requirements. We might relate to xADL [DvdHT01], that has the capability to make conceptually distinction between architectural prescription (design-time template) and architectural description (runtime state of system). However, xADL prescriptions accept a reduced degree of variability, it can specify that certain components are optional. Nearer to our goal are [IB96] with ASTER, an interconnection language for specification of application requirements. It is used to automatically build a distributed runtime system customized to meet the requirements [KI00].

We relate also with research on predictable component assembly. An important research topic in component composition is the prediction of the assembly-level properties of a component composition [HMSW02,CBS01]. Here most effort is directed toward prediction of static properties (end-to-end latency, memory consumption [FEHC02]), where the same property of an assembly can be calculated from the properties of the components, requiring no additional information. For non-quantitative properties, approaches focus on usually one property: deadlock [IT02a], reliability [SM02]. We consider mostly non-computable properties in our model. The properties of a composed component in our model are usually seen as abstract features, expressed at a higher semantic abstraction level than the properties of the parts. Having the structural constraints as part of a composable component's description specifies which properties put together and assembled will emerge the higher-level assembly property. The structural constraints are a flexible mechanism to enforce a predictable assembly of non-quantitative and non-computable properties.

7 Conclusions

In this paper we present *CCDL*, a description language for composable components in multi-flow architectures. We have introduced hierarchically composable components as a means to achieve finetuned customization of component based systems.

The goal of CCDL is to express guidelines for the component composition. CCDL descriptions can be used by automatic composition tools that implement

requirements driven compositions strategies. We have built a Composer and used it in achieving self-customizable network protocols.

A strength of our approach is that it permits a high degree of unanticipated variability, it permits to easily formulate and solve new requirements and to discover and use in given composition problems new component types, with minimal user intervention, which is very important in the case of self-customizable systems. Composable components as deployed in our model and the mechanism of defining them through their structural constraints offer the necessary flexibility, while guaranteeing a predictable assembly.

References

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[BBB+00] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Technical concepts of component-based software engineering, CMU/SEI-2000-TR-008. Technical report, Carnegie Mellon Software Engineering Institute, May 2000.

[BG97] Don Batory and Bart Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2), February 1997.

[CBS01] Proceedings of the 4th ICSE Workshop on Component-Based Software Engineering dedicated to Component Certification and System Prediction, 2001.

[DvdHT01] Eric M. Dashofy, Andre van der Hoek, and Richard N. Taylor. A highly-extensible, XML-based architecture description language. In *Proceedings of the Working IEEE/IFIP Conference on Software Architectures (WICSA 2001)*, Amsterdam, Netherlands, 2001.

[FEHC02] A.V. Fioukov, E.M. Eskenazi, D.K. Hammer, and M.R.V. Chaudron. Evaluation of static properties for component-based architectures. In *Proceedings 28th EUROMICRO conference on Component-based Software Engineering*, Dortmund, Germany, September 4th–6th 2002.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch, or, why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, pages 179–185, Seattle, Washington, April 1995.

[Gar01] David Garlan. Software architecture. In J. Marciniak, editor, *Wiley Encyclopedia of Software Engineering*. John Wiley & Sons, 2001.

[Ham02] Dieter K. Hammer. Component-based architecting for component-based systems. In Mehmet Askit, editor, *Software Architectures and Component Technology*. Kluwer, 2002.

[HMSW02] Scott A. Hissam, Gabriel A. Moreno, Judith A. Stafford, and Kurt C. Wallnau. Packaging predictable assembly. In *IFIP/ACM Working Conference on Component Deployment (CD2002)*, Berlin, Germany, June 20–21 2002.

[IB96] Valerie Issarny and Christophe Bidan. Aster: A framework for sound customization of distributed runtime systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems*, pages 586–593, Hong-Kong, May 1996.

[IS01] Paola Inverardi and S. Scriboni. Connectors synthesis for deadlock-free component based architectures. In *Proceedings of the 16th ASE*, Coronado Island, California, USA, November 2001.

[IT02a] Paola Inverardi and Massimo Tivoli. Correct and automatic assembly of COTS components: an architectural approach. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19–20 2002.

[IT02b] Paola Inverardi and Massimo Tivoli. The role of architecture in component assembly. In *Proceedings Seventh International Workshop on Component-Oriented Programmin (WCOP) at ECOOP*, Malaga, Spain, June 2002.

[KI00] Christos Kloukinas and Valerie Issarny. Automating the composition of middleware configurations. In *Automated Software Engineering*, pages 241–244, 2000.

[MT00] N. Medvidovic and R. Taylor. A classification and composition framework for software architecture description languages. *IEEE Transactions on Software Engineering*, Vol. 26 (No. 1):70–93, January 2000.

[OGT+99] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems*, 14(3):54–62, May/June 1999.

[SM02] Judith Stafford and John McGregor. Issues in predicting the reliability of composed components. In *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, Florida, USA, May 19–20 2002.

[Szy97] Clemens Szypersky. *Component Software: Beyond Object Oriented Programming*. Addison-Wesley, 1997.

[SJBV02] Ioana Şora, Nico Janssens, Yolande Berbers, and Pierre Verbaeten. A component composition model to support unanticipated customization of systems. In *Workshop on Unanticipated Software Evolution (USE) at ECOOP 2002*, Malaga, Spain, June 2002.

[SMBV02] Ioana Şora, Frank Matthijs, Yolande Berbers, and Pierre Verbaeten. Automatic composition of systems from components with anonymous dependencies. In *Proceedings of TOOLSEE 2001 - Technology of Object-Oriented Languages and Systems (TOOLS) East-Europe 2001*, Sofia, Bulgaria, March 2002.

[SVB02] Ioana Şora, Pierre Verbaeten, and Yolande Berbers. Using component composition for self-customizable systems. In I. Crnkovic, J. Stafford, and S. Larsson, editors, *Proceedings - Workshop On Component-Based Software Engineering: Composing Systems from Components*, pages 23–26, Lund, Sweden, 2002.

[Wil01] D.S. Wile. Ensuring general-purpose and domain-specific properties using architectural styles. In *CBSE4 Proceedings*, Toronto, Canada, May 2001.

[XML00] Extensible Markup Language (XML) 1.0 (second edition) W3C recommendation 6 october 2000, `http://www.w3.org/tr/rec-xml`, 2000.

A Logical Basis for the Specification of Reconfigurable Component-Based Systems*

Nazareno Aguirre** and Tom Maibaum

Department of Computer Science, King's College London
Strand, London, WC2R 2LS, United Kingdom
Tel: +44 207 848 1166, Fax: +44 207 848 2851
aguirre@dcs.kcl.ac.uk, tom@maibaum.org

Abstract. We present a logic and a prototypical specification language for specifying and reasoning about component-based systems with support for dynamic, i.e., run-time, architectural reconfiguration. We present the logic, an adaptation of an existing one proposed for specifying reactive systems, and some results that demonstrate its suitability for the specification of reconfigurable systems.
We then explicate how the specification language can be used to specify a reconfigurable (sub)system via layers defining component templates, association/connector templates and a layer specifying reconfiguration operations used to dynamically change the system architecture. We also illustrate the expressive power and proof capabilities of the logic.

1 Introduction

Due to the complexity and size of current software systems, the notion of structural architecture of systems, and its relationship to systems analysis and design, has come to play an important role in today's software development processes.

Special specification languages, called *architecture description languages* [10], were proposed to describe and analyse properties of (sometimes evolving) architectures. Many of these are able to deal with what is called dynamic reconfiguration, i.e., with the description of operations which may modify the system's structure at run time [9]. While architecture description languages (ADLs) provide constructs for modelling the architecture of a system, they often do not support reasoning about possible system evolution. In other words, some ADLs support the definition of components, interconnections and transformation rules or operations for making architectures change dynamically, but any kind of reasoning about behaviours is often performed in some "meta-language", sometimes informally. Moreover, the description of architectural elements in ADLs, particularly those related to dynamic reconfiguration, is usually done in an operational way, as opposed to declaratively [4,6,12].

* This work was partially supported by the Engineering and Physical Sciences Research Council of the UK, Grant Nr. GR/N00814.

** On leave from Departamento de Computación, FCEFQyN, Universidad Nacional de Río Cuarto, Río Cuarto, Córdoba, Argentina

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 37–51, 2003.

Being able to specify and reason about the consequences of using certain reconfiguration operations in a declarative manner would add abstraction to what, to our understanding, can be operationally specified by ADLs. We therefore propose a temporal logic as a formal basis for the specification of reconfigurable systems. Temporal logic provides a declarative and well-known language to express behavioural properties, and is currently used in several branches of software engineering.

We adapt a logic proposed by Manna and Pnueli [7] for the specification of reactive systems for this purpose. We present the logic and some results that demonstrate its suitability for the specification of dynamically reconfigurable systems. A prototypical language based on this logic is defined, where systems specifications are hierarchically organised around the following notions:

- the notion of components, which are represented by classes that define templates for these components;
- the notion of connector type, which we call associations, which are then used to define the potential ways in which components may be organised in a system;
- the notion of subsystem, the new notion that defines the (coarse grained) unit of modularity from which reconfigurable systems are built, and which conveys the information about what components, what associations and what reconfiguration operations are used to define the module.

It is not our aim to propose another architecture description language, but to study an alternative declarative and formal semantics for software architectures. We prefer to illustrate the capabilites and expressive power of the formalism by defining a simple front-end to our logic (our prototypical language). This is simpler than trying to relate, at this stage of our work, our logic to existing high-level ADLs. In addition, it allows us to show how an interesting specification mechanism, hierarchical component organisation, can be achieved using our proposed formal basis.

2 The Logic

We start by describing the logic we use as a core for the specification of reconfigurable systems. This is a variant of a logic widely used for the specification of reactive systems, Manna-Pnueli logic [7,8]. A considerable amount of work has been done on the Manna-Pnueli logic, including the development of software tools for supporting the specification of systems [2].

Most of the definitions in this section are adapted from the ones in [7] and [11]. The use of the logic for expressing properties of systems is standard. The changes to the logic consist mainly in replacing the use of local variables by the use of, what we call, flexible function symbols, and by allowing some predicates to be interpreted in a state-dependent way. We need this in order to be able to specify reconfiguration.

2.1 Syntax

An *alphabet* (sometimes called *signature* or *vocabulary*) for this logic consists of: *(i)* a set $\mathbf{S}$ of sorts, *(ii)* a set of $(\mathbf{S}^* \times \mathbf{S})$-indexed flexible function symbols, *(iii)* a set of $(\mathbf{S}^* \times \mathbf{S})$-indexed rigid function symbols, *(iv)* a set of $(\mathbf{S}^*)$-indexed flexible predicate symbols, *(v)* a set of $(\mathbf{S}^*)$-indexed rigid predicate symbols and *(vi)* a countable set of $\mathbf{S}$-indexed variables.

Typed terms are constructed from the symbols of the vocabulary as usual. Formulae are constructed also in the usual way, using the traditional propositional connectives, equality, the unary temporal operators $\bigcirc$ and $\Diamond$, the binary temporal operator $\mathcal{U}$, and quantification over variables. For the definition of the semantics of the logic we consider $\neg$ and $\rightarrow$ as the only propositional connectives, since the others can be obtained from these.

Terms are used to denote individuals, i.e., elements of the universe of discourse, and operations on them. For the specification language (and the application domain) we are interested in, we will need, for instance, terms to denote integers, strings, booleans, etc, and the usual operations on them.

The intended meaning of propositional connectives is the standard. Having in mind that validity of formulae (in a model) will be subject to a (current) state, and that states are linearly organised, the intended meanings of $\bigcirc$□ and □$\mathcal{U}$□ are "□ is true in the next state" and "□ is true (at least) until □ becomes true", respectively. This is formalised in the next section.

2.2 Semantics

First, let us introduce some definitions, necessary in order to give the semantics of this logic.

De□nition 1. *Given an alphabet $\mathcal{A}$, a (semantic) structure M for it is a mapping that assigns:*

- *for each sort S in $\mathcal{A}$, a set S_M,*
- *for each rigid function symbol $f : S_1 \ldots S_k \rightarrow S$, a function*

$$f_M : S_{1_M} \ldots S_{k_M} \rightarrow S_M$$

- *for each rigid predicate symbol $p : S_1 \ldots S_k$, a relation*

$$p_M \subseteq S_{1_M} \times \ldots \times S_{k_M}$$

Given an alphabet $\mathcal{A}$ and an $\mathcal{A}$-structure M, a *state* is a function s that maps:

- every flexible function symbol $f : S_1 \ldots S_k \rightarrow S$ to a function

$$f_M : S_{1_M} \ldots S_{k_M} \rightarrow S_M$$

- every flexible predicate symbol $p : S_1 \ldots S_k$ to a relation

$$p_M \subseteq S_{1_M} \times \ldots \times S_{k_M}$$

A *trajectory* σ in an $\mathcal{A}$-structure M is an infinite list of states. Given a trajectory $\sigma = s_0, s_1, \ldots$, we denote by $\sigma^{(k)}$ the suffix $s_k, s_{k+1}, \ldots$.

An *assignment* for an $\mathcal{A}$-structure is a mapping that, for every variable $V : S$, assigns a value $a \in S_M$. Given an assignment A, a variable $x : S$ and a value $d \in S_M$, we denote by $A_{x \lhd d}$ the assignment that coincides with A for all variables $y \neq x$, and that maps x to d.

Definition 2. *Let $\mathcal{A}$ be an alphabet. An* interpretation *is a triple $I = (M, A, \sigma)$, where M is a $\mathcal{A}$-structure, A is an assignment for M, and σ is a trajectory in M.*

Given an interpretation $I = (M, A, \sigma)$, we denote by $I_{x \lhd d}$ the interpretation $(M, A_{x \lhd d}, \sigma)$, and by $I^{(k)}$, for a natural number k, the interpretation $(M, A, \sigma^{(k)})$.

Given an interpretation $I = (M, A, \sigma)$ for $\mathcal{A}$, we define $I(t)$, for a $\mathcal{A}$-term t, as follows:

- for a constant c, $I(c) = c_M$,
- for a variable v, $I(v) = A(v)$,
- for a rigid 0-ary function symbol $f : S$, $I(f) = f_M$,
- for a flexible 0-ary function symbol $f : S$, $I(f) = \sigma_0(f)$, where σ_0 is the first state in the trajectory σ,
- for a term $f(t_1, \ldots, t_k)$, where f is a rigid function symbol,

$$I(f(t_1, \ldots, t_k)) = f_M(I(t_1), \ldots, I(t_k)),$$

- for a term $f(t_1, \ldots, t_k)$, where f is a flexible function symbol,

$$I(f(t_1, \ldots, t_k)) = \sigma_0(f)(I(t_1), \ldots, I(t_k)),$$

We are ready to define *satisfaction* of a formula under a given interpretation.

Definition 3. *Let $\mathcal{A}$ be an alphabet, and $I = (M, A, \sigma)$ an interpretation. We define satisfaction of a formula α (over alphabet $\mathcal{A}$) in I, in symbols $\overset{I}{\models} \alpha$, as follows:*

- $\overset{I}{\models} t_1 = t_2$ *if and only if* $I(t_1) = I(t_2)$,
- $\overset{I}{\models} p(t_1, \ldots, t_k)$, *for a rigid predicate p, if and only if* $(I(t_1), \ldots, I(t_k)) \in p_M$
- $\overset{I}{\models} p(t_1, \ldots, t_k)$, *for a flexible predicate p, if and only if*

$$(I(t_1), \ldots, I(t_k)) \in \sigma_0(p)$$

- $\overset{I}{\models} \neg\alpha$ *if and only if it is not the case that* $\overset{I}{\models} \alpha$,
- $\overset{I}{\models} \alpha_1 \rightarrow \alpha_2$ *if and only if* $\overset{I}{\models} \neg\alpha_1$ *or* $\overset{I}{\models} \alpha_2$,
- $\overset{I}{\models} \bigcirc\alpha$ *if and only if* $\overset{I^{(1)}}{\models} \alpha$,

$\Box \overset{I}{\models} \Diamond \Box$ *if and only if there exists a* $k \geq 0$ *such that* $\overset{I^{(k)}}{\models} \Box$,

$\Box \overset{I}{\models} \Box_1 \mathcal{U} \Box_2$ *if and only if for some* $k \geq 0$, $\overset{I^{(k)}}{\models} \Box_2$ *and for all* $0 \leq i < k$, $k \geq 0$, $\overset{I^{(i)}}{\models} \Box_1$

$\Box \overset{I}{\models} \forall x \in S : \Box$ *if and only if for all* $d \in S_M$ *it is the case that* $\overset{I_{x \lhd d}}{\models} \Box$

Given a set of formulae $\Box$ over an alphabet $\mathcal{A}$, an $\mathcal{A}$-interpretation I is called a *model* of $\Box$ if and only if $\overset{I}{\models} \Box$, for all $\Box$ in $\Box$.

Semantic Consequence. We overload the symbol $\models$, using it now for defining a relation between sets of formulae over a signature $\mathcal{A}$. Let $\mathcal{A}$ be a signature, $\Box$ and $\Box$ sets of formulae over $\mathcal{A}$. Then, we say that $\Box$ is a *semantic consequence* of $\Box$ wrt $\mathcal{A}$, in symbols $\Box \models_{\mathcal{A}} \Box$, if and only if for every interpretation I wrt $\mathcal{A}$, if I is a model of $\Box$ then it is also a model of $\Box$. We drop the subscript of $\models$ when it is clear from the context.

Proposition 1. *The binary relation* $\models$ *of semantic consequence between sets of formulae over a signature* $\mathcal{A}$ *has the following properties:*

- $\Box$ *Reflexivity:* $\Box \models \Box$
- $\Box$ *Cut: if* $\Box \cup \Box_1 \models \Box$ *and* $\Box \models \Box$, *for all* $\Box$ *in* $\Box_1$, *then* $\Box \models \Box$
- $\Box$ *Monotonicity: if* $\Box \models \Box$, *then* $\Box \cup \Box_1 \models \Box$,

for all sets $\Box$, $\Box_1$, $\Box$ *of formulae over* $\mathcal{A}$.

2.3 A Proof Calculus

A sound proof calculus for the logic can be obtained by easily adapting the axioms for the Manna-Pnueli logic presented in [11]. Inference rules presented there also preserve validity in our adaptation; however, the definition of valid substitutability has to be changed, to reflect the fact that, in our adaptation, flexible predicate and function symbols are state-dependent, which has an impact in the quantification axioms. For the reader aware of [11], these changes consist only of adapting the definition of predicate *globsub*$(t^{\frown} x^{\frown} w)$ in [11] pp. 165–161, whose intended meaning is "the replacement of x by t in formula w does not generate new occurrences of flexible symbols within the scope of temporal operators and no new ocurrences of bound variables".

The resulting proof-theoretical consequence relation satisfies reflexivity, cut and monotonicity.

2.4 Signature and Theory Morphisms

It will be necessary for us to combine different alphabets and formulae in order to be able to build specifications. For this purpose, we need the logic to satisfy certain structural properties.

Definition 4. *A* signature morphism Σ *between signatures* $\mathcal{A}$ *and* $\mathcal{B}$ *as a function that maps:*

- Σ *each sort in* $\mathbf{S}_{\mathcal{A}}$ *to a sort in* $\mathbf{S}_{\mathcal{B}}$,
- Σ *each flexible (resp. rigid) function symbol* $f : S_1 \times \ldots \times S_k \rightarrow S$ *in* $\mathcal{A}$ *to a flexible (resp. rigid) function symbol* $\Sigma(f) : \Sigma(S_1) \times \ldots \times \Sigma(S_k) \times S'_{k+1} \times \ldots \times S'_n \rightarrow \Sigma(S)$ *in* $\mathcal{B}$,
- Σ *each flexible (resp. rigid) predicate symbol* $p : S_1 \times \ldots \times S_k$ *in* $\mathcal{A}$ *to flexible (resp. rigid) predicate symbol* $\Sigma(p) : \Sigma(S_1) \times \ldots \times \Sigma(S_k) \times S'_{k+1} \times \ldots \times S'_n$ *in* $\mathcal{B}$,
- Σ *each variable* $x : S$ *in* $\mathcal{A}$ *to a variable* $\Sigma(x) : \Sigma(S)$ *in* $\mathcal{B}$.

Note that function and predicate symbols could be mapped to symbols with a greater arity. This is crucial for the way we deal with reconfiguration.

Having defined mappings of symbols from an alphabet to another, we define how to translate formulae from one alphabet to another, in a way that is useful for the purpose of specifying reconfigurable systems.

Definition 5. *Let* $\Sigma : \mathcal{A} \rightarrow \mathcal{B}$ *be a signature morphism. The function* $Gr_\Sigma : L_{\mathcal{A}} \rightarrow L_{\mathcal{B}}$, *is defined as follows: Given a formula* φ, *the formula* $Gr_\Sigma(\varphi)$ *is the result of translating the symbols in* φ *using* Σ, *placing fresh variables in the free spaces resulting from translating function or predicate symbols into others of a greater arity, and quantifying these universally.*

Example 1. Consider the formula $\Box[(\exists x \in S : p(x)) \rightarrow q]$. If a signature morphism maps S to S', $p : S$ to $p' : S' \times S''$, q to $q' : S''$, and $x : S$ to $x : S'$. Then, the formula resulting from the translation is:

$$\forall y \in S'' : [\Box[(\exists x \in S' : p(x \times y)) \rightarrow q(y)]]$$

Theorem 1. *Let* $\Sigma : \mathcal{A} \rightarrow \mathcal{B}$ *be a signature morphism, and* Γ *and* Δ *be sets of* $\mathcal{A}$*-formulae. Then,*

- $\Gamma \models_{\mathcal{A}} \Delta$ *implies* $Gr_\Sigma(\Gamma) \models_{\mathcal{B}} Gr_\Sigma(\Delta)$.
- $\Gamma \vdash_{\mathcal{A}} \Delta$ *implies* $Gr_\Sigma(\Gamma) \vdash_{\mathcal{B}} Gr_\Sigma(\Delta)$.

These results imply that this logic constitutes a Σ-institution [5], both considering semantic and proof-theoretic consequence.

3 The Language

Here we present a prototypical language, in which we make use of the logic defined in the previous section for specifying some standard elements found in ADLs. The language is inspired by the language defined in [3], and it is greatly influenced by ideas from CommUnity [12]. In particular, the definition of associations is based on the idea of coordination.

As we already indicated, it is not our aim to introduce yet another ADL. We use the language defined here just to illustrate the expressive power of the

logic, and the mechanisms applied to represent dynamic reconfiguration. It also allows us to show how our declarative setting makes possible to provide some interesting features.

We start by considering the specification of datatypes. A datatype specification is simply a theory presentation (i.e., a finite set of formulae) over a signature $\mathcal{ADT}$ with no flexible predicate or function symbols. We assume this specification contains the definition of all standard datatypes, such as integers, sequences, natural numbers, etc. In addition, we assume that a type NAME is defined, with a sufficiently large set of constants, and no operations. Let us denote this theory presentation by:

$$\mathcal{ADT} = \langle (S_{\mathcal{ADT}}, \emptyset, Fun^r_{\mathcal{ADT}}, \emptyset, Pred^r_{\mathcal{ADT}}, Var_{\mathcal{ADT}}), Ax_{\mathcal{ADT}} \rangle$$

3.1 Class Definitions

The basic building blocks of specifications in our prototypical language are components. Component templates are specified by means of *class definitions*. A class definition consists simply of: *(i)* finite sets of attributes and read variables whose type is a sort defined in $\mathcal{ADT}$ excluding NAME, *(ii)* a finite set of actions, which can have arguments typed with sorts in $\mathcal{ADT}$ excluding NAME. A class specification is equipped with a set of formulae over the signature:

$$(S_{\mathcal{ADT}} - \{\mathsf{NAME}\}, Rv \cup Att, Fun^r_{\mathcal{ADT}}, Act, Pred^r_{\mathcal{ADT}}, Var_{\mathcal{ADT}})$$

where Rv, Att and Act denote the sets of attributes, read variables and actions respectively. That is to say, we use read variables and attributes as flexible function symbols, and actions as flexible predicates, extending the vocabulary defined in the datatype specification. Axioms in the class specification are not allowed to use datatype NAME, since it will serve a special purpose later on. The purpose of the axioms of a class specification is to describe the meaning of the actions, i.e. their effect on attributes.

Example 2. Consider the class specification in Fig. 1. It is the specification of a producer; intuitively, the first formula indicates that when action *p-init*() (meant to "initialise" the component) is executed, the attribute *p-waiting* is set to F (false). The second formula says that in order to be able to perform the *produce* operation, the component must be not waiting. The third formula expresses that *produce*(x) causes the component to be waiting and the attribute *p-current* to be set to x in the next state. Formulae 4, 5 and 6 indicate how action *send*() works, calling action *dispatch*. Finally, formula 7 says that action *dispatch* can only be called by *send*, i.e., it cannot occur spontaneously.

It is worth noting that type item is a basic type; we do not use any particular features of elements of this type, we just assume is not a class type, i.e. item is defined in $\mathcal{ADT}$.

Class *Producer*
Read Variables: *ready-in* : boolean
Attributes: *p-current* : item, *p-waiting* : boolean
Actions: *produce*(x : item), *send*(), *dispatch*(x : item), *p-init*()
Axioms

1. $\Box[p\text{-}init() \rightarrow (p\text{-}waiting = \mathsf{F})]$
2. $\Box[\forall x \in \mathsf{item} : produce(x) \rightarrow (p\text{-}waiting = \mathsf{F})]$
3. $\Box[\forall x \in \mathsf{item} : produce(x) \rightarrow \bigcirc((p\text{-}waiting = \mathsf{T}) \wedge (p\text{-}current = x))]$
4. $\Box[send() \rightarrow ((ready\text{-}in = \mathsf{T}) \wedge (p\text{-}waiting = \mathsf{T}))]$
5. $\Box[send() \rightarrow dispatch(p\text{-}current)]$
6. $\Box[send() \rightarrow \bigcirc(p\text{-}waiting = \mathsf{F})]$
7. $\Box[[\exists x \in \mathsf{item} : dispatch(x)] \rightarrow send()]$

EndofClass

Fig. 1. Class specification *Producer*

Semantics of Class De nitions. A class specification is interpreted as a theory presentation, over the signature

$$(S_{\mathcal{ADT}}{}^{\varsigma}\, Rv \cup Att^{\varsigma}\, RF_{\mathcal{ADT}}{}^{\varsigma}\, P_{\mathcal{ADT}} \cup Act^{\varsigma}\, Var_{\mathcal{ADT}})^{\triangleright}$$

The axioms of the presentation are obtained by putting together: *(i)* the axioms explicitly provided for the class definition, *(ii)* the axioms of the datatype specification, *(iii)* a special (implicit) axiom, called the *locality axiom* for the specification, whose general form is:

$$\Box\left[\left(\bigvee_{g \in Act} \exists \overline{x}_g : g(\overline{x}_g)\right) \vee \left(\bigwedge_{a \in Att} \bigcirc(a) = a\right)\right]$$

where Act and Att are the sets of actions and attributes of the component, respectively. The intuitive meaning of the locality axiom, originally proposed in [3], is: "in every state it is the case that either one of the actions is executed, or all the attributes remain unchanged".

Note that read variables are not considered in the locality axiom; this is because read variables are special attributes, meant to be "entry points" used by a component to query the state of the environment; therefore, they are not controlled by the component, which implies they could change, from the point of view of the component, arbitrarily.

The inclusion of the axioms defining datatypes in the theory of a component definition justifies the following Theorem:

Theorem 2. *Given a class definition C, its corresponding theory is an extension of the theory of the datatype specification $\mathcal{ADT}$.*

3.2 Associations

Once classes have been defined, ways of making components interact can be defined. This is done by means of what we call *associations*. Associations are simply templates of *connectors*, in the sense of [1]. The syntax is very simple: An association consists of:

- a name for the association (distinct from names of other linguistic elements already defined),
- a set of participants, typed with class names,
- a set of connections, which are expressions of the form:

$$x \triangleright \square \leftrightsquigarrow y \triangleright \square$$

where x and y are participants of classes A and B respectively, and $\square$ and $\square$ are either:

- formulae in the languages of A and B respectively, or
- terms of the same sort, in the languages of A and B respectively.

The intended meaning of an association definition is that, whenever certain instances are related using an association instance, then they are forced to synchronise as the connections indicate.

The actual interpretation of associations, as special formulae, takes place at the next level of the specification, the subsystems.

Class *Consumer*
Read Variables: *ready-ext* : boolean
Attributes: *c-current* : item, *c-waiting* : boolean
Actions: *consume*(*x* : item), *extract*(*x* : item), *c-init*()
Axioms

1. $\Box[c\text{-}init() \rightarrow (c\text{-}waiting = \mathsf{T})]$
2. $\Box[\forall x \in \mathsf{item} : extract(x) \rightarrow ((c\text{-}waiting = \mathsf{T}) \wedge (ready\text{-}ext = \mathsf{T}))]$
3. $\Box[\forall x \in \mathsf{item} : extract(x) \rightarrow \bigcirc((c\text{-}waiting = \mathsf{F}) \wedge (c\text{-}current = x))]$
4. $\Box[\forall x \in \mathsf{item} : consume(x) \rightarrow \bigcirc(c\text{-}waiting = \mathsf{T})]$
5. $\Box[\forall x \in \mathsf{item} : consume(x) \rightarrow ((c\text{-}current = x) \wedge (c\text{-}waiting = \mathsf{F}))]$
6. $\Box[c\text{-}waiting = \mathsf{F} \rightarrow \Diamond(consume(c\text{-}current))]$

EndofClass

Fig. 2. Class specification *Consumer*

Example 3. Consider the class specifications in Figs. 1 and 2; we define the association in Fig. 3 with the intention of making producers and consumers interact. The intuitive meaning of the connections is that, whenever a producer p is connected to a consumer c by means of a connector *Prods-for*, then:

- the read variable *ready-in* in p is identified with the attribute *c-waiting* of c (and viceversa),
- the attribute *p-waiting* of p is identified with the read variable *ready-ext* of c (and viceversa),
- whenever $dispatch(x)$ occurs in p, $extract(x)$ also occurs (and viceversa).

Note the last axiom of the *Consumer* specification. It expresses a *liveness* condition on consumers. This shows the expressiveness of the logic, but it also shows how "low-level" the language is in its present status, since this kind of properties can be directly enforced as axioms[1].

Association *Prods-for*
Participants: $p : Producer$, $c : Consumer$
Connections:

$p.ready\text{-}in \leftrightsquigarrow c.c\text{-}waiting$
$p.p\text{-}waiting \leftrightsquigarrow c.ready\text{-}ext$
$p.dispatch(x) \leftrightsquigarrow c.extract(x)$

EndofAssociation

Fig. 3. Association specification *Prods-for*

3.3 Subsystems

This is the upper layer of the language. Once class and association specifications are given, a subsystem can be declared. Intuitively, we can describe a subsystem as a complex component, built out of instances of classes (basic components) inter-related by means of connectors, i.e., instances of associations. The key capability of a subsystem, what motivated the definition of the logic, is that it can have operations that dynamically change its architectural state.

A subsystem *Sub* then has a finite set of actions, whose arguments must be of types defined in $\mathcal{ADT}$, now including NAME. Also, as for class definitions, we use logical axioms for specifying the behaviour of a subsystem. The alphabet $\mathcal{A}_{Sub}$ over which the subsystem formulae are expressed is composed of:

[1] Liveness properties are an example of properties that might not be preserved when a component becomes part of certain systems. In our language, these properties can be expressed, and because of the semantics for subsystems, they will be "preserved" in any subsystem. When, because of some reason, such a property is not preserved, this will be reflected as an inconsistency in the theory of the corresponding subsystem. Although we agree this is certainly not the best way of "catching" the anomaly, we believe this matter can be solved by defining a more suitable "front-end" to the logic. However, we choose at the moment to illustrate the expressive power of the logic, and not to be concerned about the specification language definition, something orthogonal to the main ideas of our work.

- the sorts defined in $\mathcal{ADT}$,
- the rigid function and predicate symbols defined in $\mathcal{ADT}$,
- the flexible function and predicate symbols resulting from class definitions, adding to all of them an extra parameter of sort NAME,
- a flexible predicate symbol $A : \mathsf{NAME}$ for each class definition A,
- a flexible predicate symbol $R : \underbrace{\mathsf{NAME} \times \cdots \times \mathsf{NAME}}_{k \text{ times}}$ for each association definition R with k participants
- a flexible predicate symbol $a : S_1 \times \cdots \times S_k$ for each subsystem action of type $a(x_1 : S_1 \times \cdots \times x_k : S_k)$.

Constants of type NAME represent names of instances. Predicate $A : \mathsf{NAME}$, for a class definition A, is meant to characterise the names of live instances of type A in each state. Predicate R for an association R is used to denote the instances of connectors in each state. Clearly, there exists a signature morphism $\sigma_{A \times Sub}$ from the signature of every class definition A into the signature described above.

Example 4. Consider the subsystem specification in Fig. 4. It is a description of a complex component, built out of instances of producers and consumers interrelated by connectors of the kind defined by *Prods-for*. The subsystem consists of four operations: *(i) init*(), which is specified by axioms 1-6 (it is meant to be an initialisation operation), *(ii) change*(y), specified by axioms 7-10, makes the only producer of the subsystem, P, to produce for consumer y, *(iii) create*(y), which creates a new consumer, and *(iv) del*(y), which deletes an existing consumer. When the sort of a variable is not explicitly indicated in a quantification of a formula, we consider it to be NAME.

Operation *init*() creates a producer, P, and a consumer, C. It can be called only once, as specified by axiom 5. Operation *change* is the only one that can change the interconnections, besides the initialisation.

Note that the extra parameter of flexible symbols from component definitions is denoted using the "dot" notation from object orientation, making it more readable. As can be seen, this causes operations and attributes declared in classes to be "relativised" to a corresponding instance name. It is clear from the example how the language of the components is incorporated into the language of a subsystem. The extra parameter added to flexible symbols indicates to which instance the action or attribute corresponds to. For example, if we see axiom 4, it indicates that *init*() in the subsystem "calls" the initialisation operations of P and C, now denoted by $P.$*p-init*() and $C.$*c-init*(), respectively.

Semantics of Subsystems. As for basic components, we interpret subsystem specifications as theory presentations. A subsystem *Sub* describes a theory presentation over signature $\mathcal{A}_{Sub}$, whose axioms are:

- The formulae explicitly provided in the subsystem specification,

Subsystem Multiple_Consumers
Operations: $init()$, $change(x : \mathsf{NAME})$, $create(x : \mathsf{NAME})$, $del(x : \mathsf{NAME})$
Axioms

1. $\Box[init() \rightarrow \bigcirc(Producer(P) \wedge Consumer(C) \wedge Prods\text{-}for(P, C))]$
2. $\Box[init() \rightarrow \bigcirc(\forall x : Producer(x) \rightarrow x = P)]$
3. $\Box[init() \rightarrow \bigcirc(\forall y : Consumer(y) \rightarrow y = C)]$
4. $\Box[init() \rightarrow \bigcirc(P.p\text{-}init() \wedge C.c\text{-}init())]$
5. $\Box[init() \rightarrow \bigcirc(\Box(\neg init()))]$
6. $\Box[\neg init() \rightarrow \forall x : (Producer(x) \leftrightarrow \bigcirc(Producer(x)))]$
7. $\Box[\forall y : change(y) \rightarrow (\neg Prods\text{-}for(P, y)) \wedge \bigcirc(Prods\text{-}for(P, y))]$
8. $\Box[\forall y : change(y) \rightarrow [\exists y' : Prods\text{-}for(P, y') \rightarrow \neg \bigcirc (Prods\text{-}for(P, y'))]]$
9. $\Box[\forall y : change(y) \rightarrow P.p\text{-}waiting = \mathsf{F}]$
10. $\Box[\forall y : \neg change(y) \rightarrow (Prods\text{-}for(P, y) \leftrightarrow \bigcirc(Prods\text{-}for(P, y)))]$
11. $\Box[\forall y : create(y) \rightarrow (\neg Consumer(y)) \wedge \bigcirc(Consumer(y))]$
12. $\Box[\forall y : [(\neg Consumer(y)) \wedge \bigcirc(Consumer(y))] \rightarrow create(y) \vee init()]$
13. $\Box[\forall y : create(y) \rightarrow \bigcirc(y.c\text{-}init())]$
14. $\Box[\forall y : del(y) \rightarrow (Consumer(y)) \wedge \bigcirc(\neg Consumer(y))]$
15. $\Box[\forall y : [(Consumer(y)) \wedge \bigcirc(\neg Consumer(y))] \rightarrow del(y)]$

EndofSubsystem

Fig. 4. A subsystem specification

- The formulae corresponding to every class definition A, appropriately translated into the language of $\mathcal{A}_{Sub}$ by $Gr_{\Box_{A \lhd Sub}}$ (see Example 1, showing how a formula similar to Axiom 7 in *Producer* is translated to be incorporated in a subsystem),
- Implicit formulae characterising association definitions,
- Implicit formulae characterising general properties of subsystems.

We have chosen to have mutually exclusive sets of symbols for the component definitions (except, of course, for the symbols corresponding to the datatypes specification) to simplify the presentation.

Characterisation of Associations. We characterise association definitions by means of formulae that are incorporated in the theory of a subsystem. These formulae indicate the "type" of the arguments of the connectors, and how they communicate, according to the connections defined in the association. For our example, the formulae are:

$$\Box[\forall x \lhd y : Prods\text{-}for(x \lhd y) \rightarrow Producer(x) \wedge Consumer(y)]$$
$$\Box[\forall x \lhd y : Prods\text{-}for(x \lhd y) \rightarrow x \rhd ready\text{-}in = y \rhd c\text{-}waiting]$$
$$\Box[\forall x \lhd y : Prods\text{-}for(x \lhd y) \rightarrow x \rhd p\text{-}waiting = y \rhd ready\text{-}ext]$$
$$\Box[\forall x \lhd y : Prods\text{-}for(x \lhd y) \rightarrow \forall i : \mathsf{item} : x \rhd dispatch(i) \leftrightarrow y \rhd extract(i)]$$

General Properties of Subsystems. Besides the formulae characterising associations, there are other general properties that are specified by means of implicit

formulae in a subsystem. These formulae indicate, for example, that nothing can be at the same time an instance of two different classes, that operations of "dead" instances cannot take place, that a subsystem may evolve only by means of its own operations (locality of a subsystem), etc. We list here a few to show how they are expressed:

$$\Box[\forall x : \forall i \in \mathsf{item} : x \triangleright dispatch(i) \rightarrow Producer(x)]$$
$$\Box[\forall x : \neg(Producer(x) \wedge Consumer(x))]$$

The justification for the need of flexible predicates is their use to denote action occurrence, which should clearly be state-dependent. It is not sufficient to have flexible propositional variables, since we want to be able to deal with parameterised actions (for instance, note that when building subsystems, actions from basic components need to be relativised to instance names, which requires the consideration of a new parameter of type NAME for the corresponding predicate). For the case of function symbols, it is not sufficient to have local variables as in [7] because attributes of classes (program variables) have to be "relativised" to instance names when considered in a subsystem; therefore, flexible 0-ary functions (local variables) generated by a class A have to be represented by flexible unary functions in an including subsystem *Sub*.

The way the theory of a subsystem is constructed, importing axioms from the class specifications, and the results in Theorem 1 justify the following:

Theorem 3. *The theory corresponding to a subsystem definition Sub is an extension of the translation* $Gr_{\Box_{A \triangleleft Sub}}(\Box_A)$ *of the theory* $\Box_A$ *of every class definition A.*

Some Properties of the Subsystem. Here we provide some properties of the subsystem of Fig. 4, illustrating the expressive power of the logic. We also include a sketch of the proof of one of these properties. It is worth mentioning that these properties can be proved using our adapted version of the proof calculus for the Manna-Pnueli logic, although we cannot include the complete detailed proofs due to space restrictions.

Property 1: "After the subsystem has been initialised, P is always producing for some consumer". We can express this as follows:

$$\Box[init() \rightarrow \bigcirc(\Box(\exists y : Prods\text{-}for(P^{\triangleleft} y)))]$$

Proof: We can prove this property by using one of the proof methods for invariance described in [8]. First we prove that $init() \rightarrow \bigcirc(\exists y : Prods\text{-}for(P^{\triangleleft} y))$, which follows from Axiom 1 in the Subsystem. Then we prove that

$$init() \rightarrow \bigcirc(\exists y : Prods\text{-}for(P^{\triangleleft} y) \rightarrow \bigcirc(\exists y' : Prods\text{-}for(P^{\triangleleft} y')))$$

i.e., that after $init()$, the property is preserved by all the actions. This follows from the fact that action *change* is the only one that can "disconnect" consumers (Axiom 10 in the subsystem), and when it does, it connects P to another consumer (Axiom 7 in the subsystem), reestablishing the property.

Property 2: "After the subsystem has been initialised, if P has produced an item, the connected consumer(s) will remain to be connected to it until the item is dispatched". We express this in the following way:

$$\Box[\mathit{init}() \rightarrow \bigcirc(\Box(\exists i \in \mathsf{item} : P \triangleright \mathit{produce}(i) \rightarrow \\ \forall y : \mathit{Prods\text{-}for}(P \lessdot y) \rightarrow \mathit{Prods\text{-}for}(P \lessdot y)\, \mathcal{U}\, P \triangleright \mathit{dispatch}(i)))]$$

This property involves the combination of axioms explicitly provided in the subsystem (such as Axiom 9) with properties of the components (such as Axiom 6 in the producer specification).

Property 3: "After the subsystem has been initialised, P is the only instance of *Producer*". This is expressed as follows:

$$\Box[\mathit{init}() \rightarrow \bigcirc(\Box(\forall x : \mathit{Producer}(x) \leftrightarrow x = P))]\triangleright$$

4 Related Work

Our motivation is to formally reason about behavioural properties of dynamically reconfigurable systems. Our work is then related to approaches to the specification of software architectures, such as the work on various ADLs [1,4,9, 10].

This work is specially related to approaches to formal specificatiton of dynamic reconfiguration, such as those based on graph grammars [12] and chemical abstract machines [6]. As we indicated before, these approaches propose operational languages for specifying reconfigurable systems. Our logic provides a declarative, and therefore more abstract, framework for characterising reconfigurable systems. Moreover, reasoning can be performed within the formalism, as opposed to doing so in some meta-language, as in the mentioned alternatives.

There are many similarities between the work presented here and the work on CommUnity [12], since both are based on the logical and semantic foundations of [3]. In particular, the idea of coordination as a mechanism for achieving communication is used in our prototypical specification language for specifying associations.

The logic we presented is an adaptation of the Manna-Pnueli logic [7,8]. Unfortunately, we are not able to use the original logic for the purpose of specifying reconfigurable systems, due to a restriction on the type of flexible symbols allowed, i.e., only flexible constants/variables are allowed. Our adaptation of the logic overcomes this difficulty, but means that the extensive work on the Manna-Pnueli logic not fully applicable to our formalism. However, most of the work is relevant to our proposed adaptation. For instance, the original logic coincides with our adaptation in the fragment used for specifying datatypes and basic components; therefore, the tool support for the Manna-Pnueli logic can be used as is for assisting the verification of these parts of a specification. We believe it is not difficult to adapt the existing tool support to cope with subsystems, which require flexible function and predicate symbols. The extension also requires STeP [2] to be able to deal with structured theories and the exporting of theorems from a component to a subsystem.

5 Conclusions

We presented a logic suitable for specifying and reasoning about component-based systems with support for run-time architectural reconfiguration. The suitability of the logic is illustrated by the definition of a prototypical specification language on top of it, which shows the expressive power of the logic. The way in which associations are represented in the language allows it to express properties concerning the architecture of the system in a declarative way. Hence, operations that may change the topology of the system can be easily specified.

We think our work complements the ones regarding ADLs such as [12,1,6], by providing a uniform language to state and prove properties, that could then be related to specifications in a number of different ADLs.

Among the future work related to the logic presented here, we are studying ways of defining suitable inheritance relationships between components, which might allow us to have polymorphic reconfiguration operations in subsystems. We also want to study how specifications in some ADLs can be interpreted in our logic, to provide them with a formal logical semantics and proof capabilities.

References

1. R. Allen and D. Garlan, *Formalizing Architectural Connection*, in Proceedings of the 16th International Conference on Software Engineering ICSE '94, Sorrento, Italy, 1994.
2. N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma and T. Uribe, *Verifying Temporal Properties of Reactive Systems: a STeP Tutorial*, in Formal Methods in System Design, vol. 16, 2000.
3. J. Fiadeiro and T. Maibaum, *Temporal Theories as Modularisation Units for Concurrent System Specification*. Formal Aspects of Computing, vol. 4, No. 3, Springer-Verlag, 1992.
4. D. Garlan, R. Monroe and D. Wile, *ACME: An Architecture Description Interchange Language*, in Proc. of CASCON'97, 1997.
5. J. Goguen and R. Burstall, *Institutions: Abstract Model Theory for Specification and Programming*, in Journal of the ACM (JACM), vol. 39, No. 1, 1992.
6. P. Inverardi and A. Wolf, *Formal Specification and Analysis of Software Architetures using the Chemical Abstract Machine*, IEEE Transactions in Software Engineering, 1995.
7. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*, Springer-Verlag, 1991.
8. Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems – Safety*, Springer, 1995.
9. N. Medvidovic, *ADLs and Dynamic Architecture Changes*, in Proceedings of the Second Int. Software Architecture Workshop (ISAW-2), 1996.
10. N. Medvidovic and R. Taylor, *A Framework for Classifying and Comparing Architecture Description Languages*, In ESEC-FSE'97, 1997.
11. J. Ostroff, *Temporal Logic for Real-Time Systems*, Advanced Software Development Series, Research Studies Press Ltd, John Wiley and Sons, 1989.
12. M. Wermelinger, A. Lopes and J. Fiadeiro, *A Graph Based Architectural (Re)configuration Language*, in ESEC/FSE'01, V.Gruhn (ed), ACM Press, 2001.

An Overall System Design Approach Doing Object-Oriented Modeling to Code-Generation for Embedded Electronic Systems

Clemens Reichmann[1], Markus Kühl[2], and Klaus D. Müller-Glaser[1]

[1] Laboratory for Information Processing Technology (ITIV)
University of Karlsruhe,
Engesserstr.5, 76128 Karlsruhe, Germany
{Reichmann,Mueller-Glaser}@itiv.uni-karlsruhe.de
http://www.itiv.uni-karlsruhe.de
[2] Research Center for Information Technology,
Haid-und-Neu-Str. 10-14, 76128 Karlsruhe, Germany
Kuehl@FZI.de
http://www.fzi.de

Abstract. In this paper a new approach for an overall system design is presented. It supports object-oriented system modeling for software components in embedded systems in addition to time-discrete and time-continuous modeling concepts. Our approach provides structural and behavioral modeling with front-end tools and simulation/emulation with back-end tools. The UML metamodel is used for storing CASE data in a MOF object repository and XMI (XML Metadata Interchange format) is used to interchange this data with UML-CASE-tools. The CASE tool chain we present in this paper supports concurrent engineering including versioning and configuration management. It provides adaptors for the tools MATLAB/Simulink/Stateflow[1] and ARTiSAN Real-Time Studio[2] as well as an importer/exporter of UML/XMI. Utilizing the Unified Modeling Language notation for an overall system design cycle, the focus of this paper lies on the subsystem coupling of heterogeneous systems and on a new code generation approach.

1 Introduction

The design of embedded electronic systems has changed due to a broad introduction of the object-orientation paradigm. It is also characterized by rapidly increasing complexity and shorter product cycles today.

[1] MATLAB/Simulink/Stateflow is registered trademark of Mathworks, Inc.
[2] Real-Time Studio is registered trademark of ARTiSAN Software Tools, Inc.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 52–66, 2003.

With the introduction of the Unified Modeling Language (UML) as a standard [6] and well-accepted notation for software design, the object-orientation paradigm is more and more accepted and supported in commercial CASE tools.

These facts are results of a trend towards increasing use of more software components in typical embedded system applications. Systems in automotive applications tend to have numerous software subsystems, event-driven subsystems and often a control subsystem in the time-continuous domain. These systems usually perform tasks like diagnosis, self-calibration, encrypted data-communication, and even database-oriented system processing.

A commonly used approach to handle system complexity in the system design process is called concept-oriented rapid prototyping[3] [1,2]. Concept-oriented rapid prototyping defines a way for the fast conversion of an executable specification, which captures the requirements by using modeling techniques for state-event, time-continuous, and software modeling, into a functional prototype. This prototype mainly serves to clarify system goals and must support a widespread range of hardware interfaces. It uses automatic code generation based on CASE tools like MATLAB/Simulink, ASCET-SD[4] or Statemate[5], and powerful, general-purpose, extensible hardware. The cost of rapid prototyping hardware is not critical, as the hardware is not specific and can be reused for different prototypes.

Up to now concept-oriented rapid prototyping was influenced by tools for Computer-Aided Systems Engineering (CASE) using the state chart theory (modeling of hierarchical finite-state-machines) or control systems engineering using block diagrams. In the past much effort was spent to improve design tools for a better integration of state charts and block diagrams. Current requirements in embedded systems design shift the attention from a pure state chart and block diagram approach to a more software-oriented view, which is captured by modern CASE tools in Software Engineering domain. A major problem, common to all commercial software CASE tools, is a lack of support for control systems engineering whereas modeling of time-discrete or state-oriented systems is well supported. Therefore, development of mixed-domain systems with both time-discrete and time-continuous subsystems and additional software components is not a continuous design process today. Thus there is a need to unify the description of the model to a single notation based on one metamodel to enable an overall design technique. Furthermore the integration of the domain-specific parts in terms of code generation, data and message exchange, task distribution, model versioning, user management and automated transformation between the domains is important. In our work the designer has the possibility to model in different modeling domains (time-discrete, time-continuous, software) and notations, which are transferred to one metamodel. This is done to achieve an executable specification running on a rapid prototyping platform.

In the following section we will summarize the related work for concept-oriented rapid prototyping (see Sect. 2). We will introduce the metamodel used in our approach

[3] Not captured in this paper: architecture and implementation rapid prototyping (RP)
[4] ASCED-SD is a registered trademark of ETAS GmbH
[5] Statemate is a registered trademark of i-Logix, Inc.

in Sect. 3. Section 4 will present our universal object-oriented modeling approach for embedded electronic systems. Then, in Sect. 5 our current work to provide a subsystem coupling technique on model-layer combined with a new automatic code generation approach (see Sect. 6) will be introduced. Afterwards, *GeneralStore*, our client/server CASE tool integration platform for mixed-domains and concurrent engineering, is described. Finally, Sect. 8 discusses results and offers a conclusion of the topic.

2 Related Work

Commercial solutions that support object-oriented modeling of embedded electronic systems are rare. Today, a wide range of CASE tools for object-oriented analysis and design is available mainly supporting pure software modeling. Software CASE tools for embedded systems modeling are often new to market. Only three well known CASE tools support object-oriented analysis and design using UML notation in addition to time-discrete modeling (e.g. state charts): ARTiSAN Real-Time Studio, i-Logix Rhapsody, and Rational Rose Realtime are tools explicitly classified for this domain.

A major drawback of those CASE tools is the lack of modeling concepts for control system engineering. For an overall system design users will have to include source-code as external C-code into these CASE environments. This approach is expensive and very rigid. Communication between both modeling domains (software to time-discrete/ -continuous system parts) is done via source code coupling. In large-scale systems, changing the system architecture during the development process is very susceptible to errors. Therefore, model-based coupling of software components and time-discrete/continuous system parts is desirable.

3 Metamodel

In our approach the whole system is described as an instance of one particular metamodel in one notation. This model has to cover all three domains: time-discrete, time-continuous, and software. The Unified Modeling Language (UML) is an Object Management Group (OMG) standard [6] which we use as system notation and metamodel. It is a widely applied industry standard to model object-oriented software. The abstract syntax, well-formedness rules, Object Constraint Language (OCL), and informal semantic descriptions specify UML.

The UML specification provides XML Metadata Interchange format (XMI) [8] to enable easy interchange of metadata between modeling tools and metadata repositories in distributed heterogeneous environments. XMI integrates three key industry standards: the eXtensible Markup Language (XML), a W3C standard, the UML, and the Meta Object Facility (MOF) [7], an OMG metamodeling standard which is used to specify metamodels.

One key aspect of UML is the four layered metamodeling architecture for general purpose manipulation of metadata in distributed object repositories (see Fig. 1) which makes it suitable for our universal object-oriented modeling approach. Each layer is an abstraction of the underlying layer with the top layer (M3) at the highest abstraction level. On the M-1 layer, which is not part of the 4-layer architecture, there is the execution code of the program. The M0 layer is comprised of the information that we wish to describe (the data). This is the source code in different languages, e.g. JAVA or C++. On the model layer (M1) there is the meta-data of the M0 layer, the so-called model. Object-oriented software is typically described on the M1 layer as a UML model. The metamodel on the M2 layer consists of descriptions that define the structure and semantics of meta-data (e.g. the UML model). These are the metamodels, e.g. UML 1.3, UML 1.4, and define the language respectively notation for describing different kinds of data (M1). Finally on the M3 layer there is the meta-meta-model. MOF is used to describe meta-models and define their structure and semantic. It is an object-oriented language for defining meta-data. MOF is self-describing. In other words, MOF uses its own metamodeling constructs.

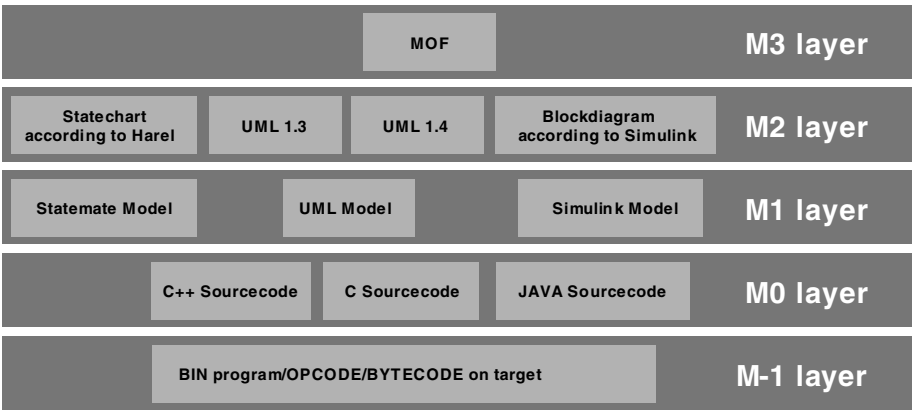


Fig. 1. 4-layer metamodeling architecture

XMI was partially influenced by the ideas for a tool-independent CASE data interchange format called CDIF [3], which was based on entity-relationship (ER) descriptions. CDIF addresses the problem of model data interchange between CASE tools.

Without a standardized interchange format for integrating more than one CASE tool, proprietary import/export filters must support the exchange of model data. In addition, new interfaces have to be implemented for tool integration. XMI is supported in a wide range of industry applications. In the machine tool domain for example, STEP, a standard for the exchange of product definition/model data, will be compatible with XMI in the future.

In the current version 1.4 of the UML standard it is possible to completely interchange model information. Nevertheless, it is not yet possible to interchange the graphical views of the model in terms of diagrams, which will be supported in the forthcoming UML 2.0.

4 Integration on Model Level

When we capture current requirements in the design process of embedded real-time systems it is necessary to handle time-continuous, time-discrete, and software design techniques. Besides the problem of using a large number of description notations/methods, an enormous amount of design data has to be handled. Another problem in many commercial CASE tools is the lack of concurrent engineering support. Only a project file based datastore is offered that can be edited only by one user at a time.

To couple different notations of the domains, transformation rules are necessary. In our approach we support different notations used by various CASE tools to model in specialized domains. The designer will choose the best tool/notation for a sub-problem and integrate the solution into the UML top-level metamodel. The software domain is modeled in UML therefore no transformation is needed. In the time-discrete domain the UML provides a notation but with the lack of well defined semantics. Here we use the semantics of David Harels concurrent hierarchical state charts [12] implemented in the CASE tool Statemate. It has an XMI interface and can interchange the time-discrete model with our integration platform *GeneralStore* (see Sect. 7).

The main difficulty when using different description domains in a complex embedded systems design is the integration of control subsystems. There are two possible solutions to overcome this problem:

1. Integration of the time-continuous subsystem using the reverse engineering mechanism of modern CASE tools for software engineering. This case is called “subsystem coupling on source code level”. One major problem here is that the control subsystem is shown as a black box subsystem encapsulated inside a class with the loss of information for other designers (see top of Fig. 2).
2. A bidirectional transformation rule (see bottom of Fig. 2) enables the synchronization of the time-continuous subsystem to a representation in the UML notation that uses simple object- and class diagrams (see left of Fig. 3). With an automated CASE tool-coupling layer, this transformation is reversible. This technique is called “subsystem coupling on model-layer”.

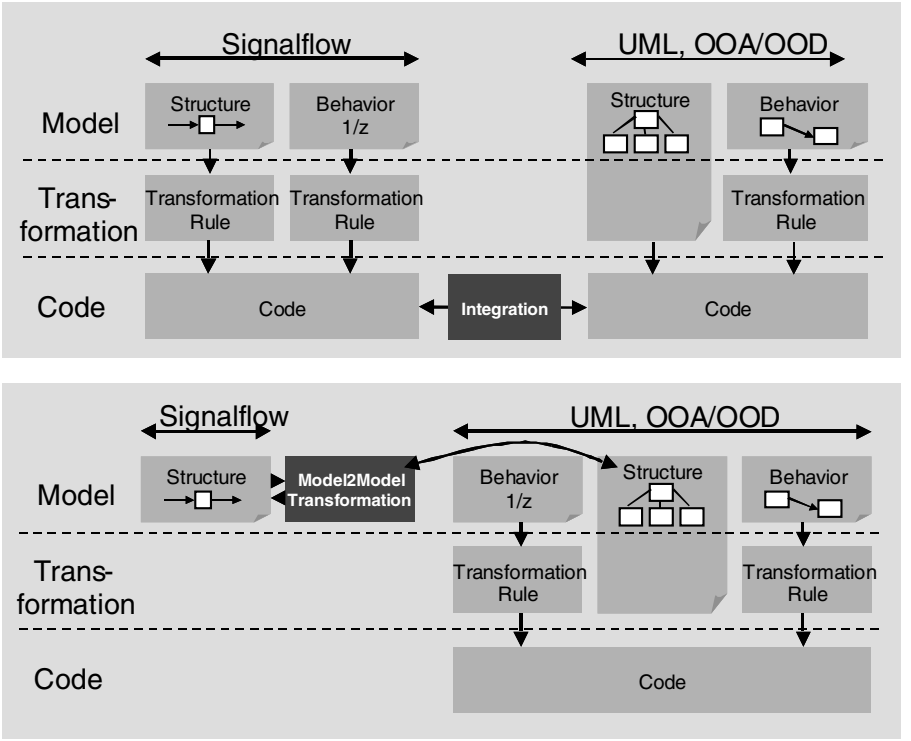


Fig. 2. Subsystem coupling on source-code level (top) and on model level (below)

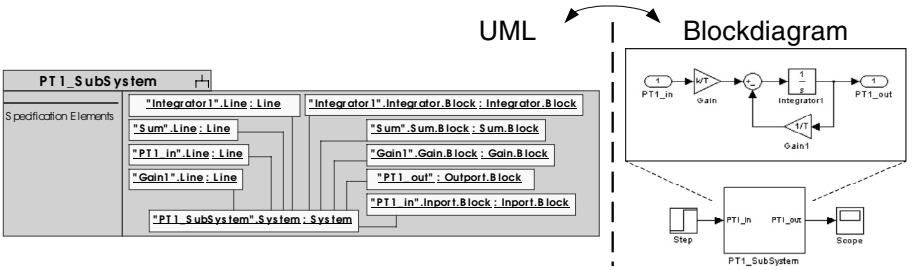


Fig. 3. Transformation of a block diagram to UML

Being more transparent the second approach is the more suitable way for a convenient integration of both modeling domains. Figure 3 shows a small footprint of the design process, which is formed by a so-called bidirectional transformation rule. Simplified, blocks in the block diagram are translated to objects in the UML metamodel. Connections represented by lines are also UML objects. The hierarchy in the block diagram is achieved by UML subsystems and links between objects. Parameters of the

blocks are slots on objects. The classifications of objects are classes whereas links are instances of associations. These elements are located in a separate UML package. This transformation rule is the result of a long-term scientific project at the Laboratory for Information Processing Technology (ITIV) at the University of Karlsruhe. We have currently developed a CASE tool integration environment (*GeneralStore*, see [4,5] and Sect. 7) that provides the necessary underlying design process support that will be introduced next.

5 Design Process

From our point of view the design process starts with object-oriented analysis using an UML CASE tool where we model use-cases to catalog the requirements. Then each requirement of the system specification is translated to a scenario modeled with a message sequence chart, state machine, or activity diagram. Non-functional requirements are modeled as OCL[6] constraints or added as comments. By doing so an initial class model is automatically introduced.

At the next step the developer arranges the class model in various class diagrams. These diagrams have to be refactored. Generalizations are identified and similar classes have to be combined. Dependencies and associations are revealed. With these steps the so-called analysis-model is achieved. This first analysis model is not in the main focus of our work but illustrate the overall process.

The model of the embedded system then has to be divided in parts, where each part is mentioned as a software component, a subsystem in the control-domain, or a state chart in the time-discrete domain (see Fig. 4).

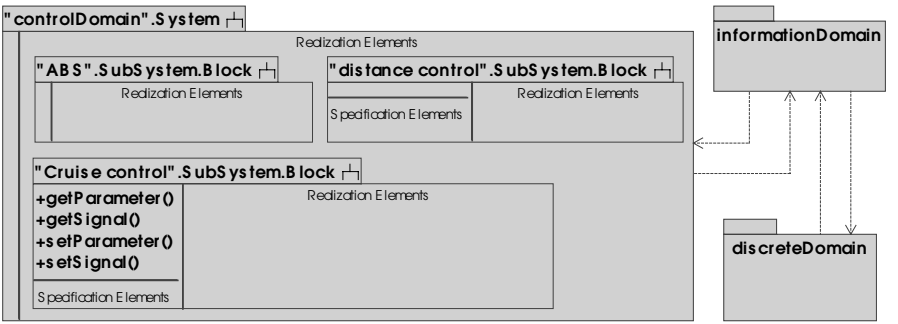


Fig. 4. Different domains as UML subsystems

[6] Object Constrain Language an OMG standard

Developers of different domains automatically select different notations and metamodels to describe the problem of the subsystem when using best fitting CASE tools. The integration platform *GeneralStore* applies automated transformations on these notations to convert them to the top-level UML metamodel (see Sect. 4).

Specialists in control system design use their notations (block diagrams) and tools (e.g. MATLAB/Simulink) to model in their domain. For integration with the overall system the control subsystem is transformed to the UML metamodel and vice versa. Thus we have a white box integration of the model in UML notation (compare Fig. 3 and Fig. 5 right).

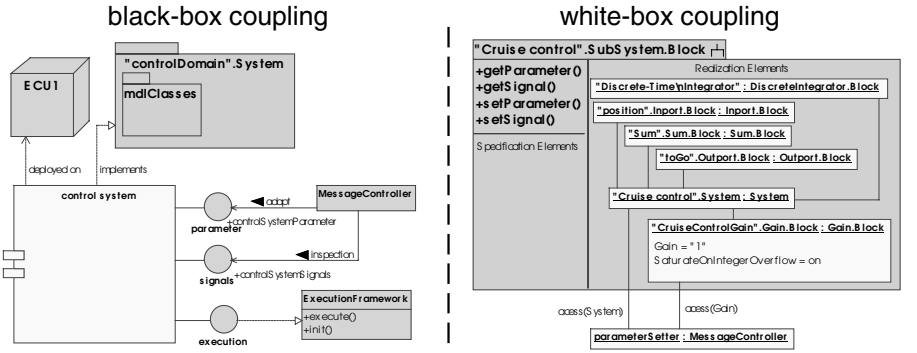


Fig. 5. Control system in UML view: black-box coupling (left) and white-box coupling (right)

Specialists in the software domain can model their subsystem and the interface to the control-domain in UML (see Fig. 5). In UML we can model the task distribution (see bottom part of Fig. 6), scheduling and message exchange of the heterogeneous system. In Fig. 5 there is the component *control system*, which is running on the electronic control unit *ECU1*. This component encapsulates the control domain, or more exactly its implementation. The *parameter* interface (see Fig. 5 left) allows manipulation to the parameters of the control system, whereas the *signals* interface provides access to the internal state of the control system, e.g., for diagnostic reasons. The *execution* interface is used for scheduling and task allocation of the control domain. The component is the black-box view of the control system.

In addition the designer can use the white box view (see Fig. 5 right) of the control system to model the access to the signals or parameters explicitly or in a generic way. On the right hand side of Fig. 5 the object *parameterSetter* has access to the parameter *Gain* of *CruiseControlGain*, which corresponds to the block *Gain* in the block diagram (see Fig. 3).

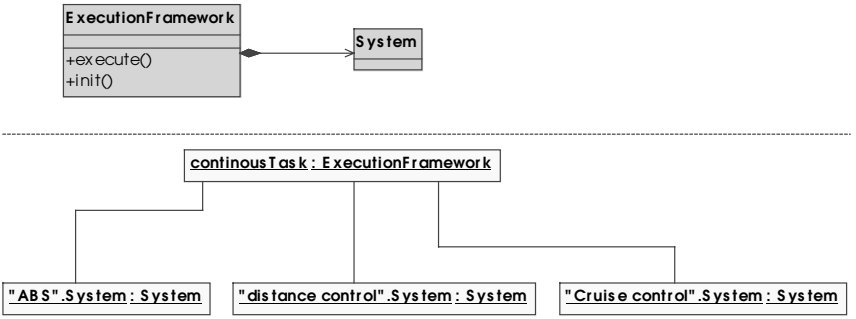


Fig. 6. Class diagram of execution modeling (top) and modeling task distribution (bottom)

The shown design pattern for domain coupling is based on an automatically generated wrapper of the control and time-discrete domain. This is done by inspection of the generated code of the supported commercial code generators. The wrapper is generated on UML model level and is transformed to code with our code generator (see Sect. 6) during code generation phase. The control domain wrapper provides access to parameters and signals. The other partitions are built with commercial generators and, as stated before, the whole system is generated to get an executable specification.

Although there are state machines in the UML metamodel, the semantic is not completely specified. But for an executable specification we absolutely need well-defined semantics of state charts. To solve this problem designers use the Harel state chart semantics [12] implemented in the CASE tool Statemate or Rhapsody in C to model the time-discrete subsystem. The coupling of the time-discrete model to the other domains is accomplished using a design pattern similar to the control-domain described above.

Now we can run the generated system on the RP[7] target. Usually, real systems will not fulfill the expected tasks completely at the first try. To find errors we use model monitoring. Test cases are modeled in UML, generated by our code generator, and run in a unit test mode on the target. After recording the whole scenario of the failed test case it can be shown in the UML model offline when the user is stepping through it.

6 Code Generation

There are highly efficient commercial code generators on the market. In safety critical systems certificated code generators have to be used to fulfill the requirements. The *GeneralStore (GS)* platform allows partitioning the whole system into subsystems to invoke different domain specific code generators.

[7] Rapid Prototyping

For control-systems there are commercial code generators like Target Link[8], Embedded Coder[9] or ECCO[10]. In the time-discrete domain we utilize the code generator of Statemate[11] (Rhapsody in MicroC). In software domain commercial code generators only generate the stubs of the static UML model while behavioral functionality has to be implemented by the software engineer.

As we focus on a completely generated executable specification it is necessary to generate code of the overall model. Therefore we provide a code generator as a *GS* plug-in to enable structural and behavioral code generation directly from a UML model. The body specification is done formally in the Method Definition Language (MeDeLa). Which is an abstract action language based on Java syntax. Our template code generator is using the Velocity engine to generate Java or C++ source code. Velocity is an Apache open source project focused on HTML code generation. It provides macros, loops, conditions, and callbacks to the data model business layer.

The different domains have interactions, e.g., signal inspection, adoption of control system parameters at runtime or sending messages between time-discrete and software artifacts. There should be one scheduler on each ECU[12] as many commercial code generators provide a specific scheduling mechanism. Integration of these different frameworks on one ECU is error prone. In our approach these design tasks could be modeled in UML with assistance of MeDeLa as outlined in Sect. 5. Afterwards our code generator renders the specified glue (interactions between domains). We provide a highly flexible modeling process, which is supported by an integration platform described in the following section.

7 Client/Server Architecture for CASE

To keep the system model manageable for designers, CASE tool integration is necessary. In [1,2] an open design environment based on CDIF for the development of mechatronic systems was presented. Many of the experiences from this project influence our current work.

After the design of prototypes during the last two years the focus of our current work is on a so-called *Integration Platform GeneralStore (GS)*. The setup of the *GS* follows a 3-tier architecture well known in the software engineering domain. On the lowest layer (database layer) a commercial object-relational database management system ORACLE[13] respectively MySQL[14] was selected. On the business-layer we

[8] From dSPACE GmbH
[9] From Mathworks, Inc.
[10] From ETAS GmbH
[11] From i-Logix, Inc.
[12] Electronic Control Unit
[13] ORACLE is registered trademark of ORACLE Corporation
[14] MySQL is an open source project: see www.msql.com

provide user authentication, transaction management, object versioning, and configuration management.

For managing CASE data, *GS* supports UML as its metamodel (see Sect. 2). *GS* uses MOF as its database scheme for storing UML artifacts. Inside the database layer an abstraction interface keeps *GS* independent from the given database.

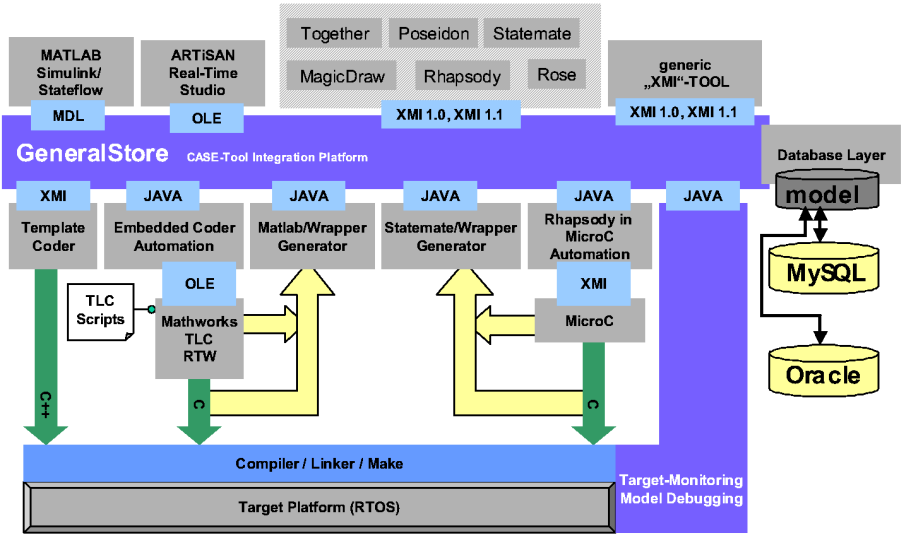


Fig. 7. Architecture of integration platform *(GeneralStore)*

On the business layer of *GS* the mediator pattern [11] is used to keep the CASE tool integration simple and its adaptor uniform. The transformations for specific notations supported by CASE tools are implemented in plug-ins (see top of Fig. 7). The code generation plug-ins (Embedded Coder and Rhapsody in MicroC) controls the transformation to the source code. Their wrapper generators are automatically building the interface in the UML model (see Fig. 7 in the middle).

While interim objects enclose MOF elements, the CASE adaptor (see class *GSTool* in Fig. 8) stays thin and highly reusable. Another reason for using interim objects on the business layer is because of object identity to handle object versioning and configuration management. To visualize these circumstances, an example is taken where a unique identification number is added to a subsystem block from the time-continuous system domain. After the block is checked out from the repository, a designer moves this subsystem to another hierarchy layer. Despite the fact that it is the same compound object, the system controller in the time-discrete system part keeps track of this link because of the subsystem identification number.

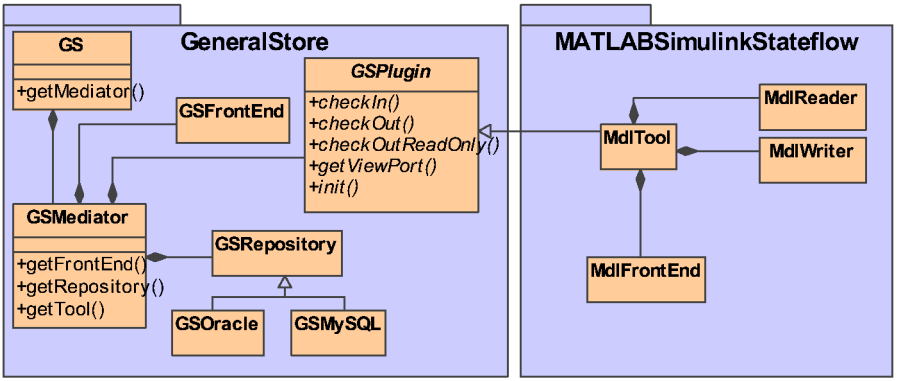


Fig. 8. Plug-in architecture of *GeneralStore*

On the presentation layer *GS* provides three principal CASE tool adapters:

- MATLAB/Simulink/Stateflow was selected for the control system design domain and the integration is done using the proprietary model file.
- Generic and specialized XMI importer/exporter filter of **.xmi*[15] files.
- And the COM[16] based integration of ARTiSAN Real-Time Studio. The tool was selected because of its focus on embedded real time systems.

Both CASE tools are bidirectional linked to the *GS* architecture. For model management and CASE tool control, *GS* offers a system hierarchy browser (compare Fig. 9). Since the internal datamodel representation of *GS* is UML, *GS* offers a system browser for all UML artifacts of the current design. Due to the large amount of MOF objects (for example: the transformed PT1 subsystem needs about 10,000 XMI entities), *GS* offers design domain specific hierarchy browsers, e.g., a system/subsystem hierarchy view for structural or time-continuous design or a package hierarchy view for software design.

8 Conclusion

The introduced universal object-oriented modeling approach supports the concurrent development of electronic systems in all design phases. We showed how heterogeneous system descriptions could work as integrated parts of an object repository based-Client/Server CASE tool environment. Based on an object diagram representation,

[15] XML file based
[16] Microsoft Component Object Model

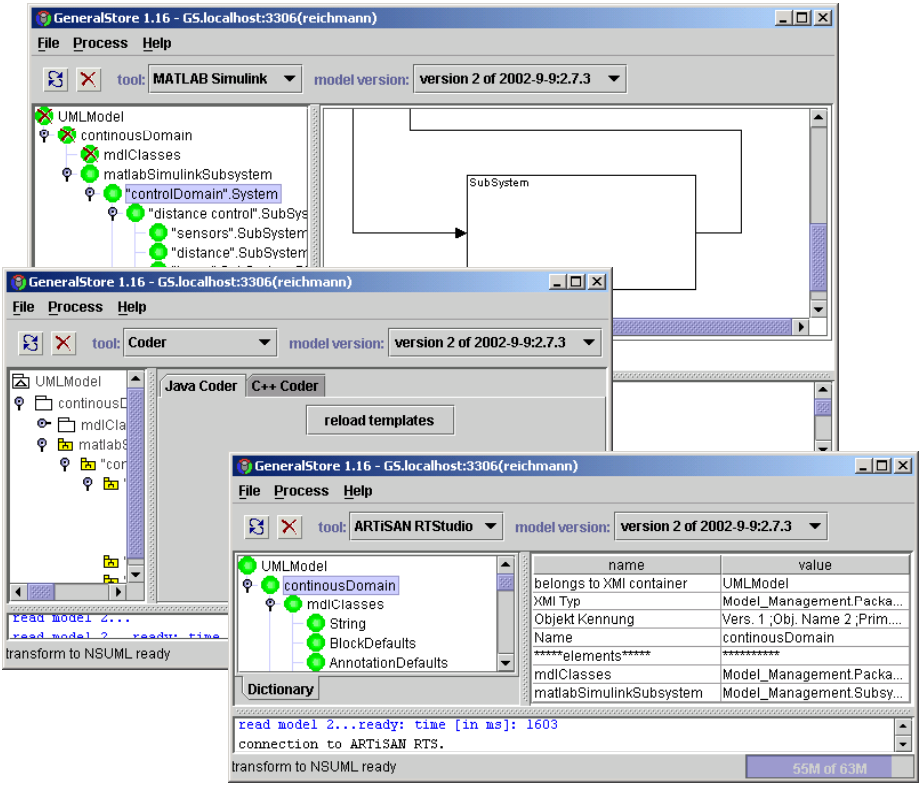


Fig. 9. *GeneralStore's* MATLAB, Coder, and ARTiSAN UML plug-in

time-continuous subsystems and software components can be modeled using one single description style. A direct linkage between different description domains is possible on an abstract model level. While transforming all subsystem parts to an uniform object notation, adding additional model information to time-continuous blocks will enable system designers to start with system simulation early in the design process.

Embedded electronic systems can be subdivided in time-discrete, time-continuous and software domain. Each domain uses its specific notation. A highly flexible design process was described to integrate those notations, which are supported by CASE tools, to one UML model. The glue between the domains is modeled in UML. Finally the overall system is transformed to source code with the assistance of commercial code generators in addition to our UML code generator.

An often-needed feature and simultaneously a drawback of project file-based CASE tools (e.g. Rhapsody, Simulink/Stateflow) is the lack of CASE tool assisted concurrent engineering. Using the presented CASE tool backend *GeneralStore* together with MATLAB/Simulink, an interim project file is created each time a designer

checks out a part of the model. This is possible at any specific model hierarchy point. The checked out subsystem hierarchy becomes protected. Other designers still have read access to the last version of this subsystem and can obtain read/write access to other subsystems in the time-continuous hierarchy.

One major drawback of using UML/MOF from XMI as a metamodel for system description is the deficiency of a standardized graphical representation for class and object diagrams. Up to now, this is one of the most requested topics for UML 2.0 and the next XMI generation. On the other side, using XMI and the UML metamodel for the description of embedded systems enables model exchange with other CASE tools. Today, at least 10 software CASE tools on the market can handle XMI descriptions from the information point of view (import of a model without graphical description).

9 Outlook

Future work has to focus on the definition of a tailored design process and the integration of CASE tools for requirements management (e.g. DOORS[17]). On the back-end side of our development environment simulation and emulation of system models is going to be supported.

We will examine the integration of further modeling tools to estimate the saved integration effort using UML/MOF. Especially for large systems of different design domains, our universal object-oriented modeling approach for the design of embedded electronic systems based on MOF will show its advantages.

Furthermore it is planned to work on enhanced modeling of operations in UML for code generation, e.g., collecting information from activity charts, object collaboration diagrams, and state charts.

It is also considered to enable modeling for reconfigureable hardware components, e.g. FPGA[18]s that means creating VHDL code from an UML model.

References

1. A. Burst; M. Wolff; M. Kühl; K.D. Müller-Glaser: A Rapid Prototyping Environment for the Concurrent Development of Mechatronic Systems, ECEC, Erlangen, Germany, 1998.
2. A. Burst; M. Wolff; M. Kühl; K.D. Müller-Glaser: Using CDIF for Concept-Oriented Rapid Prototyping of Electronic Systems, RSP, Leuven, Belgium, 1998.
3. EIA / CDIF Technical Committee: CDIF / CASE Data Interchange Format. EIA Interim Std. EIS / IS-106–112, 1994.
4. M. Kühl; C. Reichmann; B. Spitzer; K.D. Müller-Glaser: Universal Object-Oriented Modeling for Rapid Prototyping of Embedded Electronic Systems, RSP, Monterey, USA, 2001.

[17] DOORS is registered trademark of Quality System & Software, Inc.
[18] FPGA: field programmable logic array

5. M. Kühl; C. Reichmann; K.D. Müller-Glaser: Universal Object-Oriented Modeling with ARTiSAN Rts and MATLAB/Simulink, ARTiSAN User Conference 2001, London, UK, 2001.
6. Object Management Group: OMG / Unified Modeling Language (UML) V1.4, 2001.
7. Object Management Group: OMG / Meta Object Facility (MOF) V1.4, 2001.
8. Object Management Group: OMG / XML Metadata Inter-change (XMI) V1.0, 2000.
9. B.P. Douglass: Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns. Addison-Wesley, 1999.
10. M. Fowler: Refactoring - Improving the Design of Existing Code. Addison-Wesley, 1999.
11. E. Gamma et al.: Design Patterns – elements of reusable object-oriented software; Addison-Wesley, 1994.
12. David Harel: Statecharts: a visual formalism for complex systems. Science of Computer Programming 8 (1987,3), 231–274

Composing Specifications of Event Based Applications*

Pascal Fenkam, Harald Gall, and Mehdi Jazayeri

Technical University of Vienna, Distributed Systems Group
A-1040 Vienna, Argentinierstrasse 8/184-1
{p.fenkam,h.gall,m.jazayeri}@infosys.tuwien.ac.at

Abstract. The event based architectural style has been recognized as fostering the development of large-scale and complex systems by loosely coupling their components. It is therefore increasingly deployed in various environments such as middleware for mobile computing, message oriented middleware, integration frameworks, communication standards, and commercial toolkits. The development of applications based on this paradigm is, however, performed in such an ad-hoc manner that it is often difficult to reason about their correctness. This is partly due to the lack of suitable specification and verification techniques. In this paper, we review the existing theory of specifying and verifying such applications, argue that it cannot be applied for the development of large-scale and complex systems, and finally propose a novel approach (*LECAP*) for the construction of correct event based applications. Our approach is superior to the existing approaches in many respects: 1) we assume a while-parallel language with a synchronization construct, 2) neither a pending event infrastructure nor a consume statement are required, 3) a dynamic (instead of static) binding is assumed, 4) no restriction is made on the number of simultaneous executions of the same program 5) our approach is oriented towards top-down development of systems. The paper also presents two examples for illustrating the approach.

1 Introduction

The increasing complexity of (distributed) software systems has led to the investigation of new methods that can ease their development. Such methods include formal methods, object orientation, extreme programming, component based software engineering. These approaches are supported by emerging paradigms such as objects, encapsulation, polymorphism, concurrency, communication, shared variables, or mobile agents. Each of these paradigms comes with its set of features that challenge existing specification and verification techniques.

* This work was supported by the European Commission in the Framework of the IST Program, Key Action II on New Methods of Work and eCommerce. Project number: IST-1999-11400 MOTION (MObile Teamwork Infrastructure for Organizations Networking).

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 67–86, 2003.

The event based architectural style is one such paradigm. Essentially, an event based (EB) system consists of allowing some components *called subscribers* to express their interests in some kind of information, while allowing other components *called publishers* to publish this information. The EB system is responsible for matching publications to subscriptions and forwarding them to interested subscribers. An application that includes such subscribers and publishers is called an *event based application*. The importance of the EB paradigm is witnessed by the increasing number of domains and tools in which it is exploited. Examples of such domains/tools are programming environments (e.g. Smalltalk), operating systems (e.g. AppleEvents [3]), communication middleware (e.g. Corba [14], Siena [7], JEDI [9], Elvin [29]), integration frameworks (e.g. OLE [5], JavaBeans [26], FIELD [25], SunSoft [28], Polylith [24], ISIS [4], Yeast [19]), and message oriented middleware (e.g. see [21]).

While considerable effort has gone into techniques for composition of software based on procedure invocation [10,15], shared data [8,22], and message passing [6,20], there is no established method for developing correct applications based on the EB paradigm. As a result, applications based on this paradigm are often developed in an ad-hoc and informal manner. Although it is not expected that all developers carry out formal techniques throughout the whole design and implementation process, they may use the intuition that has been built up during the development of the supporting techniques [12]. Further, formal techniques can be applied in relaxed versions as lightweight formal methods [2,6].

This paper proposes a novel formal approach for building correct applications using the EB paradigm. This approach is called *LECAP*: **L**ogic of **E**vent **C**onsumption **A**nd **P**ublication. This logic is compositional; hence, intrinsically oriented towards construction of complex systems. LECAP is based on Jones's rely/guarantee [17,27,32] program derivation technique which is extended in two respects. First, we extend the specification of a program to include announcement-conditions (ann-conditions). These are conditions that specify which events a program is allowed to announce. Next, we provide a rule for composing separately developed specifications into one large specification.

Let us assume that we want to build a software system that satisfies the requirements $\square_1 \lll \ldots \lll \square_n$. Our methodology consists of four steps:

1. Identify components necessary for constructing the system.
2. Develop the formal specifications $S_1 \lll \ldots \lll S_m$ of these components and verify some local properties of these specifications. These formal specifications consist in pre-conditions, post-conditions, rely-conditions, guarantee-conditions (guar-condition), wait-conditions, and ann-conditions.
3. Compute the formal specification S of the whole system using the specifications $S_1 \lll \ldots \lll S_m$ and our composition rule. The specification S is computed such that any application which refines it satisfies the requirements $\square_1 \lll \ldots \lll \square_n$.
4. Separately refine the specifications $S_1 \lll \ldots \lll S_m$ to some implementations $I_1 \lll \ldots \lll I_m$.

It is important to stress that the development of $I_1 \lhd\!\rhd\!\rhd \ldots \lhd\!\rhd\!\rhd I_m$ can be performed by different teams that know nothing about each other. Each of them simply receives some specification S_i and is required to deliver some code that satisfies this specification. In other words, $I_1 \lhd\!\rhd\!\rhd \ldots \lhd\!\rhd\!\rhd I_m$ might be off-the-shelf components that satisfy the requirements $S_1 \lhd\!\rhd\!\rhd \ldots \lhd\!\rhd\!\rhd S_m$. Indeed, this is one of the expected benefits of the loose coupling of components.

The contribution of the paper is following:

1. We provide a formal definition of a language that supports development of event based applications. This language is called *the LECAP programming language*. The presentation of this language is important since it provides some added value compared to the specification provided in [12,11]. In particular, the LECAP programming language is a parallel programming language with synchronization constructs while that of Dingel et al. [12,11] is a sequential while-language. Further, the LECAP programming language doesn't require constructs such as the "consume" statement introduced in [12,11].
2. We provide a formal specification and semantics of an event based system. We believe that this specification is simpler compared to [12,11]: no pending event infrastructure is required. The capability of delaying events is obtained naturally from the programming language through the synchronization construct.
3. A method for the specification of event based applications is provided. Besides the pre-, post-, rely-, guar-, and wait-conditions, we introduce the ann-conditions that specify the kind of events a component is allowed to announce.
4. We present a rule for composing specifications (of components) into specification of systems.
5. We present two examples that illustrate the application of the technique.

The remainder of the paper is organized as follows. The next section (Sect. 2) presents related approaches. Section 3 provides the formal definition of the LECAP programming language. Section 4 presents a logic of specifying event based programs and a rule for composing these specifications. Section 5 presents some discussions. Section 6 presents two examples that illustrate our approach and Section 7 concludes the paper.

2 Related Work

Although the event based paradigm is at the heart of countless software systems, not much work has been presented on building correct applications using this paradigm. There are three main research areas that are related to our work.

The first related area concerns event broadcasting. A significant amount of work has been done on this topic that led to a number of theories such as calculi of broadcasting systems. Examples of such calculi are the $b\Box$-calculus [13] and the CBS [23] (calculus of broadcasting systems). The issue in such works is

how to achieve fault tolerance through replication. All the components in such a system are interested in all events. The requirements are thus different from that of event based systems where each component specifies the kind of events it is interested in.

The second related area of research concerns construction of parallel programs. Jones's rely/guarantee [17] (extended e.g. by Stolen [27] and Xu [32]) and the work of Owicki/Gries [22] are among the approaches that have influenced this area. Our work is strongly based on these two works. We extend the concept of rely/guarantee specification technique through ann-conditions. We also borrow auxiliary variables from Owicki/Gries and Stolen [27] to formulate ann-conditions.

The third area of work is about verifying the correctness of event based applications. The only work we are aware of is by Dingel et al. [12,11]. A method for reasoning about event based applications is proposed. This approach, which we call Dingel's approach is also based on Jones's rely/guarantee paradigm. To illustrate the contribution of the paper, we give a summary of Dingel's approach and some of its shortcomings.

Let $S = (M^{\frown} V^{\frown} EM^{\frown} Ex)$ denotes a system consisting of a set of methods M, a set of global variables V, a binding of methods to events EM, and a set of external events Ex. Further, a specification is a 4-tuple $(P^{\frown} R^{\frown} G^{\frown} Q)$, where $P^{\frown} R^{\frown} G^{\frown} Q$ denote the pre-, rely-, guar-, and post-conditions. To show that the system S satisfies some partial correctness property T, 4 steps are required:

1. Define the pre-, rely, and post-conditions of the system: $P^{\frown} R^{\frown} Q$.
2. For each method $m \in M$, define the guarantee conditions G_m and $G_{M\setminus\{m\}}$ such that
   $(m^{\frown} V^{\frown} EM^{\frown} Ex)$ and $(M \setminus \{m\}^{\frown} V^{\frown} EM^{\frown} Ex)$ satisfy $(P^{\frown} R \vee G_{M\setminus\{m\}}{}^{\frown} G_m{}^{\frown} Q)$
3. Conclude using rely/guarantee soundness that $(M^{\frown} V^{\frown} EM^{\frown} Ex)$ satisfies $(P^{\frown} R^{\frown} \bigvee_{m\in M} G_m{}^{\frown} Q)$
4. Show that any system that satisfies $(P^{\frown} R^{\frown} \bigvee_{m\in M} G_m{}^{\frown} Q)$ also satisfies T.

This approach has a number of shortcomings.

1. It assumes a programming language with a *consume* construct. Each method must start with this statement that specifies which events the method is interested in. Dingel et al. use the consume construct to model invocation of methods by the EB system and to trace changes in the pending event infrastructure. They, however, recognize that this construct "introduces an unnecessary dependency between the event-method binding and the program of a method.[12,11]" Further, no real programming language or event based system needs such a construct.
2. The underlying specification technique is based on a pending event infrastructure. The primary intent of an EB system is not to queue events, but to dispatch them to subscribers. Queuing events results from the fact that an EB system might not be able to forward events at the speed at which they are received. Hence, we suggest that, although it may be important to take it into consideration at the implementation level, a mechanism for queuing

events should only influence the abstract model in a such a way that it does not complicate the reasoning too much.

3. Dingel et. [12,11] assume in their work that when a program is running it cannot be triggered anymore. No mechanism is however given for achieving this. On the other hand there are applications where such a limitation is not acceptable (e.g. reservation systems).
4. The approach doesn't take the definition of new subscriptions into consideration. A static binding EM is assumed. In this sense, the approach seems to miss a fundamental aspect of the event based paradigm which is (because of loose coupling) to ease the integration of new components.
5. Dingel's approach is intended for a-posteriori verification of systems instead of stepwise construction of systems: components of the completed programs are verified in isolation and then put together where general properties are proved. Jones [17] argues that such approaches are unacceptable as program development methods: erroneous design decisions taken in early steps are propagated until the system is implemented and proven.
   Although Dingel et al. [12,11] do not claim to propose a method for the stepwise construction of systems, the fact that their approach is based on Jones's rely/guarantee method for the construction of interfering programs may lead one to expect that it can also be used for such a purpose. To see why this is difficult, let us consider the following development method naively derived from the above reasoning technique:
   To construct a system S that satisfies some partial correctness property T, 6 steps must be followed:
   a) Define the pre-, rely, and post-conditions $P \,\text{‹}\, R \,\text{‹}\, Q$ of the system.
   b) Identify the set of methods M of the system.
   c) For each method $m \in M$ (not yet implemented), define the guarantee conditions G_m and $G_{M\setminus\{m\}}$ such that $(m \,\text{‹}\, V \,\text{‹}\, EM \,\text{‹}\, Ex)$ and $(M \setminus \{m\} \,\text{‹}\, V \,\text{‹}\, EM \,\text{‹}\, Ex)$ satisfy $(P \,\text{‹}\, R \vee G_{M\setminus\{m\}} \,\text{‹}\, G_m \,\text{‹}\, Q)$
   d) Conclude that $(M \,\text{‹}\, V \,\text{‹}\, EM \,\text{‹}\, Ex)$ satisfies $(P \,\text{‹}\, R \,\text{‹}\, \bigvee_{m\in M} G_m \,\text{‹}\, Q)$
   e) Show that any system that satisfies $(P \,\text{‹}\, R \,\text{‹}\, \bigvee_{m\in M} G_m \,\text{‹}\, Q)$ also satisfies T.
   f) Now, refine each method m to some implementation.
   This approach, however, doesn't work since there is nothing that relates the specifications $(P \,\text{‹}\, R \vee G_{M\setminus\{m\}} \,\text{‹}\, G_m \,\text{‹}\, Q)$ of the different methods to each other. This relation should be provided by the event based system. The methods should communicate with each other through the event based system by announcing and consuming events. This notion of announcement and consumption of events is, however, absent from the specification, hence the insufficiency of the specification and the inadequacy of the approach.

We propose an approach that overcomes these shortcomings.

3 The LECAP Programming Language

This section introduces the LECAP programming language. This language is used for the development of while-parallel programs with shared variables. The

particularly of LECAP programs is that they may communicate through an event based system. We define the syntax of the language and its operational semantics. We also give a definition of the concept of event based systems.

3.1 Syntax

A LECAP program is a while-program augmented with parallel, synchronization, and event publication constructs. Its syntax can be defined as follows:

$P ::= x := e \mid P_1; P_2 \mid$ **if** b **then** P_1 **else** P_2 **fi** $\mid$ **while** b **do** P **od** $\mid$ **await** b **then** P **end** $\mid P_1 \| P_2 \mid$ **announce(e)** $\mid$ **skip**.

Three constructs in this language need some explanations: the parallel construct, the await construct, and the announce construct. The first models nondeterministic interleaving of the atomic actions of P_1 and P_2. Synchronization and mutual exclusion are achieved using the await construct. The announce construct allows announcement of events. It is intended for the notification of the EB system which in turn triggers some other programs. The execution of announce is limited to sending the event to the EB system. The sequential composition $P_1; P_2$ can thus be defined in the usual way as a program that first behaves as P_1 and follows as P_2 if P_1 terminates. To simplify the deduction rules, it is required that variables used in the boolean test cannot be accessed by programs running in parallel. This constraint can be removed as discussed in [27]. We say that a program z_0 is a subprogram of another program z iff z can be written in one the following forms:

- $z_1; z_0; z_2$;
- **if** b **then** z_1 **else** z_2 **fi**, with z_0 a subprogram of z_1 or z_2;
- **while** b **do** z_1 **od**, with z_0 a subprogram of z_1;
- $z_1 \| z_2$, with z_0 a subprogram of z_1 or z_2;
- **await** b **do** z_1 **od** , with z_0 a subprogram of z_1.

3.2 Event Based System

Although there are various paradigms that make up an event based system in practice, not all of them are needed at the abstract level. We construct an abstract model based on a set of programs, a set of events, a binding, and a set of variables. An event is a piece of data that may be published by a program. Subscriptions are templates for allowing categorization of events. The set of programs is the set of handlers of events. Such programs are triggered when an event is announced that matches one of their subscriptions. The programs in an event based systems may not only communicate (by announcing and consuming events), but they may also share some variables.

Definition 1. *An event based system is a 4-tuple* $(\mathcal{E}, \mathcal{M}, \square, \mathcal{B})$ *composed of a set of events* $\mathcal{E}$*, a set of programs* $\mathcal{M}$*, a set of global variables* $\square$ *shared among programs in* $\mathcal{M}$*, and a binding* $\mathcal{B}$ *which maps each program to its set of subscriptions.*

The behavior of an EB system consists of providing facilities for announcing and receiving events. Programs announce events that are dispatched to some other programs. The purpose of an event is thus to trigger some programs. The process of determining which programs are interested in an event is called matching. A matching is performed between an event and a subscription. This is a query that describes the interest of a program in receiving some events. Such subscriptions are typically assertions that characterize events. They can be viewed as total functions defined on the set of events and returning true or false. An example of subscription is: $x : x$ *starts with 'John'*. An example of event that matches this subscription is: *'John is leaving the office'*.

Definition 2. *Assuming an EB system* $(\mathcal{E}, \mathcal{M}, \mathcal{B})$*, a subscription* s *is a total function from* $\mathcal{E}$ *to* $\{True, False\}$.

$\mathcal{B}(z)$ denotes the set of subscriptions of a program z and determines the set of events z is interested in.

Definition 3. *Given an event* e*, we define* $subscribers(e) = \{z \in \mathcal{M} \mid \exists s \in \mathcal{B}(z), s(e) = true\}$ *as the set of programs that are interested in the event* e.

Among the set of events $\mathcal{E}$, there may be some external events. These are events that are announced by programs not in $\mathcal{M}$. We denote the set of external events as $\mathcal{E}_x$ and the set of programs that subscribed to some of these events as $\mathcal{M}_x$. Formally, $\mathcal{M}_x = \bigcup_{e \in \mathcal{E}_x} subscribers(e)$.

3.3 Operational Semantics

We give the operational semantics of the LECAP programming language in the style of [1]. A state maps all programming variables to values and a configuration is a pair $\langle p, s \rangle$ where p is a program and s is a state. The semantics of the LECAP programming language is given relative to an EB system $(\mathcal{E}, \mathcal{M}, \mathcal{B})$. In the sequel z denotes a program that is different from the empty program $\square$.

An environment transition $\xrightarrow{v}$ is the least binary relation on configurations such that:

- $\langle z, s_1 \rangle \xrightarrow{v} \langle z, s_2 \rangle$, environment transitions are only allowed to modify the state of EB systems.

A program transition $\xrightarrow{i}$ is the least binary relation on configurations such that one of the following holds:

- $\langle skip, s \rangle \xrightarrow{i} \langle \square, s \rangle$,
- $\langle u := r, s \rangle \xrightarrow{i} \langle \square, s[u \triangleleft r] \rangle$, where $s[u \triangleleft r]$ denotes the state obtained from s by mapping the variable u to the value r and leaving all other state variables unchanged,

- $\langle announce(e); z^{\epsilon}\ s\rangle \xrightarrow{i} \langle \| subscribers(e) \cup \{z\}^{\epsilon}\ s\rangle$. The effect of announcing an event is to trigger the set of programs that subscribed to this event and execute them in parallel with the remainder of the announcing program. Programs triggered by an event are part of the running program and their transitions are therefore internal transitions.

- $\langle z_1; z_2^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_2^{\epsilon}\ s_2\rangle$ if $\langle z_1^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s_2\rangle$ and **announce(e)** is not a subprogram of z_1,

- $\langle z_1; z_2^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_3; z_2^{\epsilon}\ s_2\rangle$ if $\langle z_1^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_3^{\epsilon}\ s_2\rangle$, $z_3 \neq \square$ and **announce(e)** is not a subprogram of z_1,

- $\langle$if b then z_1 else $z_2^{\epsilon}\ s\rangle \xrightarrow{i} \langle z_1^{\epsilon}\ s\rangle$ if $s \models b$,

- $\langle$if b then z_1 else $z_2^{\epsilon}\ s\rangle \xrightarrow{i} \langle z_2^{\epsilon}\ s\rangle$ if $s \models \neg b$,

- $\langle$while b do z od$^{\epsilon}\ s\rangle \xrightarrow{i} \langle z;$ while b do z od$^{\epsilon}\ s\rangle$ if $s \models b$,

- $\langle$while b do z od$^{\epsilon}\ s\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s\rangle$ if $s \models \neg b$,

- $\langle \{z_1 \parallel z_2\}^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_2^{\epsilon}\ s_2\rangle$ if $\langle z_1^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s_2\rangle$,

- $\langle \{z_1 \parallel z_2\}^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_1^{\epsilon}\ s_2\rangle$ if $\langle z_2^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s_2\rangle$,

- $\langle \{z_1 \parallel z_2\}^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \{z_3 \parallel z_2\}^{\epsilon}\ s_2\rangle$ if $\langle z_1^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_3^{\epsilon}\ s_2\rangle$, $z_3 \neq \square$.

- $\langle \{z_1 \parallel z_2\}^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \{z_1 \parallel z_3\}^{\epsilon}\ s_2\rangle$ if $\langle z_2^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle z_3^{\epsilon}\ s_2\rangle$, $z_3 \neq \square$.

- $\langle$ await b do z_1 od $^{\epsilon}\ s_1\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s_n\rangle$ if $s_1 \models b$, and there exists a list of configurations $\langle z_2^{\epsilon}\, s_2\rangle$,...,$\langle z_{n-1}^{\epsilon}\, s_{n-1}\rangle$, such that $\langle z_{n-1}^{\epsilon}\, s_{n-1}\rangle \xrightarrow{i} \langle \square^{\epsilon}\ s_n\rangle$ and for all $1 < k < n^{\epsilon}\, \langle z_{k-1}^{\epsilon}\, s_{k-1}\rangle \xrightarrow{i} \langle z_k^{\epsilon}\ s_k\rangle$,

The meaning of an await statement is not very clear when its body does not terminate [31]. When it however terminates the final state is required to satisfy the post-condition. Given that we are not interested (in this work) in non-terminating programs we can stipulate that any computation of an await-statement has a finite length.

Besides the state of the system that programs may read and update, they also have local variables that are hidden such that environment transitions are not allowed to access them. We do not model this concept in our work since it has no impact on our rules.

Definition 4. *A configuration c_1 is disabled if there is no configuration c_2 such that $c_1 \xrightarrow{i} c_2$.*

Definition 5. *A computation is a possibly infinite sequence of environment and program transitions:* $\langle z_1, s_1\rangle \xrightarrow{l_1} \ldots \xrightarrow{l_{k-1}} \langle z_k, s_k\rangle \xrightarrow{l_k} \ggg$ *such that the final configuration is disabled if the sequence is finite. A computation is blocked if it is finite and the program of the last computation is different from* $\Box$*. A computation terminates iff it is finite and the program of the last configuration is* $\Box$*.*

The above operational semantics doesn't explicitly discuss the case of events announced by the environment (including external events). The programs triggered by these events are part of the environment and their transitions are environment transitions.

Given a computation $\Box$, then $Z(\Box)$, $S(\Box)$ and $L(\Box)$ are the projections to sequences of programs, states and transition labels, while $Z(\Box_k)$, $S(\Box_k)$ and $L(\Box_k)$ and $\Box_k$ denote respectively the k'th program, the k'th state, the k'th transition label and the k'th configuration. The number of configurations in $\Box$ is denoted $len(\Box)$. If $\Box$ is infinite, then $len(\Box) = \infty$.

In the sequel, $cp[z]$ denotes the set of computations $\Box$ such that $Z(\Box_1) = z$. These computations are called computations of z.

4 Specification Language

We show how rely- and guar- conditions can be extended and used for the specification of LECAP programs. These programs are parallel programs that might use synchronization constructs and are based on the event paradigm. Specifically, we show that the quintessence of the logic of specified programs proposed by Stolen [27] can be reused.

In the style of VDM [18], hooked variables are used to denote an earlier state. For any variable v of type $\mathcal{T}$, there exists a corresponding hooked variable $\overset{\Box}{v}$ of type $\mathcal{T}$. Hooked variables cannot appear in programs.

An assertion is a boolean formula on states. Let P be an assertion and s_1 and s_2 be two states. We write $(s_1, s_2) \models P$ if P is true when each hooked variable v in P is assigned the value $s_1(v)$ and each unhooked variable is assigned the value $s_2(v)$. If P has no hooked variable, it may be thought of as the set of all states, such that $s \models P$. We write $\models P$ if P is valid in the actual structure.

4.1 Specification

If $(\mathcal{L}, \mathcal{M}, \Box, \mathcal{B})$ is an EB system, a specification is of the form $(\mathcal{L}, \mathcal{M}, \Box, \mathcal{B}) :: (P, R, W, G, E, A)$, where the *pre-condition* P and the *wait-condition* W are unary assertions while the *rely-condition* R, the *guar-condition* G, and the *post-condition* E are binary assertions. The ann-condition A is a set of assertions defined on computations. It characterizes the kind of events a program may announce.

Definition 6. *Given a computation* $\Box$*, an assertion* Q*, and an event* e*,* Q *conditions the announcement of* e *in* $\Box$ *(or* $\Box$ *satisfies* $Q \prec e$*) iff for any state* s

found along that satisfies Q there is a program transition in the remainder of of one of the following forms:

- $\langle announce(e); z^{\epsilon}\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\}^{\epsilon}\ s\rangle$.
- $\langle announce(e) \| z^{\epsilon}\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\}^{\epsilon}\ s\rangle$.
- $\langle z \| announce(e)^{\epsilon}\ s\rangle \xrightarrow{i} \langle \| subscribers(e)\ \cup \{z\}^{\epsilon}\ s\rangle$.

There should be exactly one such configuration.

De nition 7. *An ann-condition is a set of formula of the form $Q \prec e$. A computation satisfies an ann-condition $A = \{Q_i \prec e_i\}_i$ iff it satisfies each $Q_i \prec e_i \in A$*

The restriction on the number announcements of an event e along a computation can be surmounted by labelling events.

If X is a set of variables and s_1, s_2 are two states, then $s_1 \overset{X}{=} s_2$ signifies that for all variables x in X, $s_1(x) = s_2(x)$ while $s_1 \overset{X}{\neq} s_2$ means that there exists x in X such that $s_1(x) \neq s_2(x)$.

De nition 8. *Given an EB system $(\mathcal{L}^{\epsilon}\mathcal{M}^{\epsilon}\ ^{\epsilon}\mathcal{B})$, a pre-condition P, and a rely-condition R, then $ext[^{\epsilon}P^{\epsilon}R]$ denotes the set of computations such that:*

- $S(_1) \models P$,
- *for all $1 \leq j < len(\ )$, if $L(_j) = e$ and $S(_j) \overset{}{\neq} S(_{j+1})$ then $(S(_j)^{\epsilon}S(_{j+1})) \models R$.*

The previous definition characterizes computations that satisfy the pre-condition and are subject to environment transitions. Informally, 1) the initial state must satisfy the pre-condition, and 2) any environment transition which changes the global state must satisfy the rely-condition.
A specification also characterizes commitments of the implementations:

De nition 9. *Assuming an event based system $(\mathcal{L}^{\epsilon}\mathcal{M}^{\epsilon}\ ^{\epsilon}\mathcal{B})$, a guar-condition G, and a post-condition E, a wait-condition W, and an ann-condition A then $int[^{\epsilon}G^{\epsilon}E^{\epsilon}W^{\epsilon}A]$ denotes the set of computations such that:*

- *for all $1 \leq j < len(\ )$, if $L(_j) = i$ and $S(_j) \overset{}{\neq} S(_{j+1})$ then $(S(_j)^{\epsilon}S(_{j+1})) \models G$,*
- *if $Z(_{len(\ )}) = $ then $(S(_1)^{\epsilon}S(_{len(\ )})) \models E$,*
- *if $Z(_{len(\ )}) \neq $ then $Z(_{len(\ )}) \models W$*
- *$len(\ ) \neq \infty$*
- * satisfies the ann-condition A.*

The above definitions implicitly take into consideration the case of a program z_e triggered by an event e announced by z. The triggered program z_e is in fact part of the running program which becomes $z_e \| z_1$ where z_1 is the remainder of z (after the the announcement). However, in the parallel composition $z_e \| z_1$, z_e and z_1 are part of the environment of each other. They are therefore required to satisfy the rely-condition of each other. An interference free composition must hence require that they coexist.

4.2 Judgments

Definition 10. *Given an event based system* $(\mathcal{L}, \mathcal{M}, \square, \mathcal{B})$*, a judgment is a pair consisting of a program* $z \in \mathcal{M}$ *and a specification* $(\mathcal{L}, \mathcal{M}, \square, \mathcal{B}) :: (\mathcal{P}, \mathcal{R}, \mathcal{G}, \mathcal{E}, \mathcal{W}, \mathcal{A})$*. Such a judgment is denoted* $z \models (\mathcal{L}, \mathcal{M}, \square, \mathcal{B}) :: (P, R, G, E, W, A)$*.*

Definition 11. *Let us assume an event based system* $(\mathcal{L}, \mathcal{M}, \square, \mathcal{B})$*; A judgment* $z \models (\mathcal{L}, \mathcal{M}, \square, \mathcal{B}) :: (P, R, G, E, W, A)$ *is valid iff* $cp[z] \cap ext[\square, P, R] \subseteq int[\square, G, E, W, A]$ *.*

We extend the concept of judgment to the whole event based system and say that $\models (\mathcal{L}, \mathcal{M}, \square, \mathcal{B}) :: (P, R, G, E, W, A)$ is valid iff the judgment $z \models (\mathcal{L}, \mathcal{M}, \square, \mathcal{B}) :: (P, R, G, E, W, A)$ is valid for any program $z \in \mathcal{M}$.

4.3 Composition

This section aims at formulating the rule for the composition of LECAP specifications. A LECAP program that satisfies its specification is called a correct program (w.r.t. its specification).

In the remainder, if $\mathcal{B}$ is a binding, z a program, and e an event, $\mathcal{B}\dagger\{z \mapsto \neg e\}$ represents the binding obtained from $\mathcal{B}$ by subscribing the program z to the event e. By extension of this notation, if $\mathcal{S}$ is an EB system, $\mathcal{S}\dagger\{z \mapsto \neg e\}$ represents $\mathcal{S}$ with its binding $\mathcal{B}$ replaced by $\mathcal{B}\dagger\{z \mapsto \neg e\}$. $I_{\square}$ denotes the assertion $\bigwedge_{x \in \square} x = \overset{\square}{x}$. $I_{\square}$ is thus an assertion that states that the value of no variable in $\square$ changed. If A, B, and C are some binary assertions, $B \mid C$ denotes the assertion characterizing the relational composition of B and C i.e. $(s_1, s_3) \models B \mid C$ iff there exists s_2 such that $(s_1, s_2) \models B$ and $(s_2, s_3) \models C$. B^* denotes the transitive closure of B. A^B denotes an assertion that characterizes any state that can be reached from a state A by a finite number of B steps. $\mathcal{S}$ represents the event based system $(\mathcal{L}, \mathcal{M} \cup \{z_1, z_2\}, \square, \mathcal{B})$ and $\mathcal{S}_0$ represents the event based system $(\mathcal{L}, \mathcal{M}, \square, \{\})$ with an empty binding. This means that no program in $\mathcal{S}_0$ is subscribed to any event. Announcing an event has an effect neither on the state of the system nor on running programs. In this case, the ann-condition A in a specification such as (P, R, G, E, W, A) is not relevant (denoted $\perp$). The set of deduction rules proposed by Stolen [27] is thus applicable. We illustrate the consequence, the parallel and the await rules below. They are further used for the construction of our composition rule.

Consequence rule:

$$\frac{\begin{array}{l} P_2 \Rightarrow P_1 \\ R_2 \Rightarrow R_1 \\ W_1 \Rightarrow W_2 \\ G_1 \Rightarrow G_2 \\ E_1 \Rightarrow E_2 \\ z \models \mathcal{S}_0 :: (P_1, R_1, G_1, E_1, W_1, A_1) \end{array}}{z \models \mathcal{S}_0 :: (P_2, R_2, G_2, E_2, W_2, \perp)}$$

Parallel rule:

$$\frac{\begin{array}{l} \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ G_1 \Rightarrow R_2 \\ G_2 \Rightarrow R_1 \\ z_1 \models \mathcal{S}_0 :: (P, R_1, W \vee W_1, G_1, E_1, A_1) \\ z_2 \models \mathcal{S}_0 :: (P, R_2, W \vee W_2, G_2, E_2, A_2) \end{array}}{\{z_1 \| z_2\} \models \mathcal{S}_0 :: (P, R_1 \wedge R_2, W, G_1 \vee G_2, E_1 \wedge E_2, \perp)}$$

The consequence rule allows refinement of specifications by strengthening the assumptions and weake-ning the commitments.

The parallel rule ensures parallelization of programs. The program $z_1 \| z_2$ can be derived from z_1 and z_2 if they satisfy the above specifications. An important requirement for z_1 and z_2 to coexist is that the guar-condition of one implies the rely-condition of the other. Further, not both processes should be in a waiting status as well as if the execution of one of them is completed, the other should not be waiting.

Sequential Rule

$$\frac{\begin{array}{l}P_1 \wedge E_1 \Rightarrow P_2 \\ z_1 \models \mathcal{S}_0 :: (P_1 \,\text{‹}\, R \,\text{‹}\, W \,\text{‹}\, G \,\text{‹}\, E_1 \,\text{‹}\, A_1) \\ z_2 \models \mathcal{S}_0 :: (P_2 \,\text{‹}\, R \,\text{‹}\, W \,\text{‹}\, G \,\text{‹}\, E_2 \,\text{‹}\, A_2)\end{array}}{z_1; z_2 \models \mathcal{S}_0 :: (P_1 \,\text{‹}\, R \,\text{‹}\, W \,\text{‹}\, G \,\text{‹}\, E_1 | E_2 \,\text{‹}\, \bot)}$$

Await Rule:

$$\frac{z \models \mathcal{S}_0 :: (P^R \wedge b \,\text{‹}\, \text{false,false,true} \;, (G \vee I_{\square}) \wedge E \,\text{‹}\, A)}{\text{await } b \text{ do } z \text{ od} \models \mathcal{S}_0 :: (P \,\text{‹}\, R \,\text{‹}\, P^R \wedge \neg b \,\text{‹}\, G \,\text{‹}\, R^* | E | R^* \,\text{‹}\, \bot)}$$

The sequential rule is quite similar to that of sequential programming. It essentially requires that the post-condition of the first program implies the pre-condition of the second. The resulting specification is that of a program whose computations start in a state satisfying the first pre-condition, ends in a state satisfying the second post-condition, and is such that it contains a state satisfying the first post-condition.

The intent of the await-rule is to allow programs to synchronize on resources. Assume we want to construct a synchronizing program that satisfies $(P \,\text{‹}\, R \,\text{‹}\, P^R \wedge \neg b \,\text{‹}\, G \,\text{‹}\, R^* | E | R^*)$. This program needs to block when b is false. Hence, it executes when b is true. However, if the await-program executes when b is true, the await-body needs to have b as conjunct in its pre-condition. Further, if the await-program has P as pre-condition, the await-body needs to have P^R as conjunct in its pre-condition. The reason for this is that while the program is waiting for b to be true, some environment transitions may be performed that modify the state of the system in a way that satisfies the rely condition. The pre-condition of the await-body is thus $P^R \wedge b$. The rely- and wait-conditions of the wait-body results from the fact that we want the program to be executed in an atomic step (without any interference). Since the program is executed in an atomic step, if we want the await-program to guarantee G, the post-condition of its unique step (the await-body) needs to either leave all variables unchanged or satisfy G. Of course, the post-condition of the await-body also needs to satisfy E.

We now consider the above rules; still with an empty binding, but taking ann-conditions into consideration. The computations of rely- guar-, pre-, post- and wait-conditions remain the same as above. In the following, Q_1, Q_2 and Q designate some assertions, while e_i are events. $Fr(Q)$ denotes the set of free variables in the assertion Q.

Parallel Rule

$$\frac{\begin{array}{l}Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ e_i \neq e_j \,\text{‹}\quad i \in [1 \,\text{‹}\, n] \,\text{‹}\quad j \in [n+1 \,\text{‹}\, m] \\ \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge E_1) \wedge \neg(W_1 \wedge E_2) \\ G_2 \Rightarrow R_1 \\ G_1 \Rightarrow R_2 \\ z_1 \models \mathcal{S} :: (P \,\text{‹}\, R_1 \,\text{‹}\, W \vee W_1 \,\text{‹}\, G_1 \,\text{‹}\, E_1 \,\text{‹}\, \{Q_i \prec e_i\}_1^n) \\ z_2 \models \mathcal{S} :: (P \,\text{‹}\, R_2 \,\text{‹}\, W \vee W_2 \,\text{‹}\, G_2 \,\text{‹}\, E_2 \,\text{‹}\, \{Q_j \prec e_j\}_{n+1}^m)\end{array}}{\{z_1 \| z_2\} \models \mathcal{S} :: (P \,\text{‹}\, R_1 \wedge R_2 \,\text{‹}\, W \,\text{‹}\, G_1 \vee G_2 \,\text{‹}\, E_1 \wedge E_2 \,\text{‹}\, \{Q_i \prec e_i\}_1^m)}$$

Sequential Rule

$$\frac{\begin{array}{l} Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ e_i \neq e_j \text{‹} \quad i \in [1 \text{‹} n] \text{‹} \quad j \in [n+1 \text{‹} m] \\ P_1 \wedge E_1 \Rightarrow P_2 \\ z_1 \models \mathcal{S} :: (P_1 \text{‹} R \text{‹} W \text{‹} G \text{‹} E_1 \text{‹} \{Q_i \prec e_i\}_{n+1}^{m}) \\ z_2 \models \mathcal{S} :: (P_2 \text{‹} R \text{‹} W \text{‹} G \text{‹} E_2 \text{‹} \{Q_i \prec e_i\}_{1}^{n}) \end{array}}{z_1; z_2 \models \mathcal{S} :: (P_1 \text{‹} R \text{‹} W \text{‹} G \text{‹} E_1 | E_2 \text{‹} \{Q_i \prec e_i\}_1^m)}$$

If Q_1 (resp. Q_2) conditions the announcement of e_1 (resp. e_2) in z_1 (resp. z_2), the parallel composition of z_1 and z_2 is such that Q_1 (resp. Q_2) conditions the announcement of e_2 (resp. e_1) in $z_1 \| z_2$. There are two reasons why this is true:

- no event is announced by both programs. If an event was announced by both programs, the parallel composition would yield a program that announces the same event twice. This is not compatible with the definition of $Q \prec e$.
- the set of variables in any assertion in the ann-condition of z_1 is disjoint from the global state and from the set of variables of any assertion in the ann-condition of z_2 (and vice-versa). To understand the necessity of this restriction, assume a program z_1 that satisfies $x > 2 \prec e_1$ and some of its computations $\square_1$ have a configurations such that its state satisfies $x > 2$ holds. By the definition of announcement conditions there is an announcement transition in $\square_1$. On the other hand there may be a program z_2 such that when composing in parallel with z_1 there is no state any more that satisfies $x > 2$. The resulting program $z_1 \| z_2$ announces a program although the condition $x > 2$ is not satisfied. This doesn't happen when the assertions are based on auxiliary variables that appear only in one program.

The enhancement of the await-rule is trivial: if the announcement of an event is conditioned by Q in z, this event remains conditioned by Q when we embed z in an await construct.

Await rule:

$$\frac{z \models \mathcal{S}_0 :: (P^R \wedge b \text{‹ false,false,true },(G \vee I_{\square}) \wedge E \text{‹} \{Q_i \prec e_i\}_1^m)}{\text{await } b \text{ do } z \text{ od} \models \mathcal{S}_0 :: (P \text{‹} R \text{‹} P^R \wedge \neg b \text{‹} G \text{‹} R^* | E | R^* \text{‹} \{Q_i \prec e_i\}_1^m)}$$

Starting from an empty binding, we now need to successively add subscriptions to the EB system. Before investigating the composition rule, we give one more definition. Let us denote the set of events that a program possibly announces as *events*(z); let also the set $\square(z)$ =*subscribers(events*(z)) be the set of programs subscribed on the events that the program z announces. $\square^*(z)$ denotes the transitive closure of $\square$ defined as $\square(z) \cup \bigcup_{s \in \square(z)} \square^*(s)$

De nition 12. *The binding $\mathcal{B}$ of an EB system ($\mathcal{L}$‹ $\mathcal{M}$‹ $\square$‹ $\mathcal{B}$) is well founded iff for any program z, $z \not\in \square^*(z)$. A well founded binding will be denoted wf $\mathcal{B}$.*

The intent of the above definition is to avoid infinite loops in EB systems. The simplest such case is when a program subscribes to the events that itself announces. This restriction seems to be strong: a program may subscribe on events it

announces without producing infinite loops. We doubt on the necessity of such configurations and exclude them as may complicate the composition rule.

Computing the specification of a system out of those of its components consists of successively adding new subscriptions to the EB system. The process starts with a system with an empty binding. After adding a new subscription, the composition rule is applied and new specifications are derived. The algorithm is following:

```
To subscribe program z2 to  event e do:
  add  the entry (z2 , e) to the the binding;
    for each program z that announces e, do:
        apply the composition rule
        if necessary, update any specification that depends on z
```

The composition rule can now be given. For each subscription that is performed, it is required that the binding remains well founded. The claim of this rule is that if the programs z and z_2 are specified as shown in the premises while z_2 is not interested in e_1, subscribing z_2 to e_1 results in a program that behaves like z first and eventually (from a state satisfying I^{R_1}), behaves like $z_2 \| z$. This is a natural consequence of the semantics of event announcement.

Composition rule:

$$\frac{\begin{array}{l} Fr(Q_i) \cap Fr(Q_j) = Fr(Q_i) \cap \square = Fr(Q_j) \cap \square = \emptyset \\ \neg(W_1 \wedge W_2) \wedge \neg(W_2 \wedge I|E_1) \wedge \neg(W_1 \wedge E_2) \\ e_i \neq e_j \text{‹} \quad i \in [1 \text{‹} n] \text{‹} \ j \in [n+1 \text{‹} m] \\ \text{wf } \mathcal{B}\dagger\{z_2 \mapsto e_1\} \\ \square \\ P \wedge I|R^* \Rightarrow P_2 \wedge Q_1 \\ z \not\in \text{subscribers}(e_1) \\ y \models \mathcal{S} :: (P \text{‹} R_1 \text{‹} W_1 \text{‹} G_1 \text{‹} I|E_1 \text{‹} \{Q_i \prec e_i\}_1^n) \\ z_2 \models \mathcal{S} :: (P_2 \text{‹} R_2 \text{‹} W_2 \text{‹} G_2 \text{‹} E_2 \text{‹} \{Q_i \prec e_i\}_{n+1}^m) \end{array}}{z \models \mathcal{S}\dagger\{z_2 \mapsto e_1\} :: (P \text{‹} R_1 \wedge R_2 \text{‹} W_1 \text{‹} G_1 \vee G_2 \text{‹} I|E_1 \wedge P_2|E_2 \text{‹} \{Q_i \prec e_i\}_1^m)}$$

4.4 Cause of Events

Before subscribing a program to an event, one needs to know the meaning of this event. The cause of an event is an assertion that characterizes the announcement of an event in the whole system. An event is announced in the system iff this assertion is true. Let $(\mathcal{M} \text{‹} \mathcal{L} \text{‹} \square \text{‹} \mathcal{B})$ be an EB system and A_z be the ann-condition of a program z in $\mathcal{M}$. We further denote the union of all ann-conditions as $\mathcal{A} =$. The following formula gives a formal definition:

$$cause(e) = \bigvee_{Q \in S} Q \text{‹ where } S = \{Q \text{‹} Q \prec e' \in \bigcup_{z \in \mathcal{M}} A_z\} \triangleright$$

The formula says that the cause of an event is the disjunction of the assertions Q that condition the announcement of events with the same semantics as e. The formula is not applicable to events that may be announced by the environment since the environment may be non-deterministic.

5 Discussion

An important issue is the tractability of our logic. Let us consider a system in which z, and y announce the events e_1, and e_2 respectively while y is subscribed to e_1. If we further subscribe z_3 to e_2, y needs to be re-computed. Worst, all specifications that depend on y (in our case only z) need also be updated. In real project, the chain may be long and the composition can become painful. Fortunately, the difference between the different programs are syntactically clear. For instance after subscribing z_3 to y only a conjunct of the form $Q^P|T$ needs to be added to specifications that depend on y. Updating the specifications is thus a copy-paste process that can be easily mechanized. We believe that CASE tools for composing specifications can help solve this problem.

Another factor that influences the tractability of our approach is the order in which subscriptions are performed. It is obvious that if z_3 is subscribed to e_2 before y is subscribed to e_1, the changes are less significant. Thus, an initial step in composing the specification of a system out of those of its components is to find a suitable sequence of application of the composition rule.

This issue of tractability is not specific to our composition approach. Techniques of composition based on procedure invocation have comparable problems. The manifestation of such problems in practice is e.g. regression testing that tackles the issue of detecting which part of a system must be tested following the modification in another part of the system.

We argued at the beginning of this document that our approach does not require a pending event infrastructure. The question is thus, how to model such a requirement since it might be important in some cases to show e.g. that the result of an operation doesn't depend on the ordering of events. This can be done in our approach using the synchronization construct. Queuing or delaying an event until a condition Q is fulfilled means embedding the subscribed programs in an await construct conditioned by Q.

Our composition rule requires that bindings be well-founded. There are, however, cases where a program announces an event, and waits for another program to consume the event and send a result back. A typical such scenario is to simulate method invocation using the event based paradigm, which would make "caller" less strongly coupled to the "callee". To achieve this, we need an auxiliary program p and an auxiliary variable v. The purpose of p is simply to store the event it receives in v. Now, for achieving our method invocation, the caller first subscribes the auxiliary program p to events it would like to wait for. Next, the caller announces the event containing the parameters of the call and blocks (by means of the await construct). On the other side the callee is triggered by the event based system. After processing the event, the callee publishes an event to which the auxiliary program p is subscribed for which the caller is waiting. Once the auxiliary program is triggered it stores the received event in the related auxiliary variable such that the wait-condition of the caller now holds. The caller can continue its execution by reading the content of the auxiliary variable. We are working on generalizing this solution to make the various details transparent to the designers.

6 Examples

We consider an example similar to that of Dingel et al. [12,11]. The goal is to develop a system consisting in a buffer and a counter. Each time an element is added to the buffer, the counter must be incremented. Similarly, each time an element is removed from the buffer, the counter has to be decremented. We adopt two approaches for designing this system.

6.1 Example 1

We design a system with four programs. The first program (*add*) adds elements of type $\mathcal{T}$ to the buffer while the second program *incr* increments a counter *Count*. Similarly, *remove* removes elements from the buffer and *decr* decrements the counter. The global state is thus composed of *Buf* and *Count*.

Let us construct the event based system $\mathcal{S}_0 = (\mathcal{L}, \mathcal{M}, \square, \mathcal{B})$ first. We already have $\square = \{ Buf, Count\}$ and $\mathcal{M} = \{add, incr, remove, decr\}$. We deduce the empty binding $\mathcal{B} = \{add \mapsto \emptyset, incr \mapsto \emptyset, decr \mapsto \emptyset, remove \mapsto \emptyset\}$. We further extend the event based system with the set of auxiliary variables $\square_a = \{ Buf_{a1}, Buf_{a2}\}$ used in the ann-conditions. An event is a tuple consisting of an identification number and an element of type $\mathcal{T}$. The set of event is thus $\mathcal{E} = \{(id, elt) | id \in \mathbb{N} \wedge elt \in \mathcal{T}\}$. We access the identifier (resp. element) of an event *evt* using the notation $evt{\bowtie}id$ (resp. $evt{\triangleright}elt$). A subscription is a total function defined on the set of events. An example of subscription (using the lambda notation) is: $s = \square e : e{\bowtie}id > 40$

The next step in the example is to give the formal specification of the programs *add* and *incr* denoted as S_a and S_i. In this example, we want each program to run alone à-la sequential programming. The rely-conditions of the programs are false while their wait-, and guar-conditions are true. If we assume *Buf* to be an unbounded buffer, then elements can always be inserted, hence the pre-condition is true. The post-condition can be defined as $Buf = e{\triangleright}elt + \overset{\square}{Buf}$. Each program that is started is done so with an event as input. We want the program *add* to announce the event $(1, e{\triangleright}elt)$ whenever an element is added in the buffer. The ann-condition is thus $\{ Buf_{a1} = \{e{\triangleright}elt\} \prec (1, e{\triangleright}elt)\}$. For this ann-condition to be satisfied when an element is added to the buffer the conjuncts $Buf_{a1} = \emptyset$ and $Buf_{a1} = \{e{\triangleright}elt\}$ must be added to the pre- and post-conditions respectively. S_a is thus expressed as: $S_a = \mathcal{S} :: (Buf_{a1} = \emptyset, false, false, false, Buf_{a1} = \{e{\triangleright}elt\} \wedge Buf = e{\triangleright}elt + \overset{\square}{Buf}, \{ Buf_{a1} = \{e{\triangleright}elt\} \prec (1, e{\triangleright}elt)\})$.
Similarly, we define the following specification of *incr*: $S_i = \mathcal{S}_0 :: (true, false, false, false, Count = \overset{\square}{Count} + 1, \emptyset)$. Using the consequence rule we can refine it by strengthening the pre-condition: $S_i = \mathcal{S}_0 :: (Buf = e{\triangleright}elt + \overset{\square}{Buf}, false, false, false, Count = \overset{\square}{Count} + 1, \emptyset)$.

We now subscribe *incr* to events with identifier equals to 1. This is done with the subscription query $\square x : x{\bowtie}id = 1$. Applying the composition rule we obtain the new definition of S_i: $S_a = \mathcal{S} :: (Buf_{a1} = \emptyset, false, false, false, Buf_{a1} = \{e{\triangleright}elt\} \wedge Buf = e{\triangleright}elt + \overset{\square}{Buf} \wedge Count = \overset{\square}{Count} + 1, \{ Buf_{a1} = \{e{\triangleright}elt\} \prec (1, e{\triangleright}elt)\})$.

A similar reasoning yields the following specifications S_r, and S_d of *remove* and *decr* respectively:

$S_r = \mathcal{S} :: (\ Buf_{a1} = \{e \triangleright elt\} \wedge e \triangleright elt \in Buf \lhd false \lhd false \lhd false \lhd Buf_{a2} = \emptyset \wedge Buf = \overset{\square}{Buf} - e \triangleright elt \lhd \{\ Buf_{a2} = \emptyset \prec (2 \lhd e \triangleright elt)\})$,
$S_r = \mathcal{S} :: (\ Count > 0 \lhd false \lhd false \lhd false \lhd Count = \overset{\square}{Count} - 1 \lhd \emptyset)$.
Subscribing *decr* to events with identifiers equal to 2 (subscription query $\square x : x \bowtie id = 2$) and applying the composition rule (strengthening the pre-condition of *decr* first) yields the following specification:
$S_r = \mathcal{S} :: (\ Buf_{a2} = \{e \triangleright elt\} \wedge e \triangleright elt \in Buf \wedge Count > 0 \lhd false \lhd false \lhd false \lhd Buf_{a2} = \emptyset \wedge Buf = \overset{\square}{Buf} - e \triangleright elt \wedge Count = \overset{\square}{Count} - 1 \lhd \{\ Buf_{a2} = \emptyset \prec (2 \lhd e \triangleright elt)\})$.

We now specify the kind of interactions we expect from the environment: 1) the environment may not publish events such that the identifier is equal to 1 or 2. This would lead to invalid incrementation/decrementation of the counter. 2) the environment may announce any other event. Various techniques can be applied for implementing these constraints on the environment. An example of such techniques is access control.

At this stage, the system is almost useless: the programs *add* and *remove* are not accessible to the environment. We subscribe them to the events with identifiers equal to 3 ($\square x : x \bowtie id = 3$) and to events with identifiers equal to 4 respectively. The environment can thus remove elements from or add elements to the buffer as it wills. However, since the programs *add* and *remove* do not allow interference, any event that is announced while one of these programs is running will simply be ignored.

Various properties of the system can be proved based on this specification. For instance, one can verify that each computation of the system conserves the property $\#\ Buf = Count$. This is, if the number of elements in the buffer is equals to the value of the counter before a computation, this will also be the case when the computation is completed. After proving some desirable properties of the system, the different components can be implemented such that they are correct w.r.t. their specifications.

6.2 Example 2

In the previous example, if an event arrives while a program is running, the event will simply be discarded. We extend the previous example such that if an event arrives while a program is running, the triggered program has to wait until it is save to run.

We consider the same system $\mathcal{S}_0$ as in the previous example. We enrich the global state with a boolean variable called *sentinel*. We want *add* to be of the form: `await` *sentinel* `do` *addbody*. A specification of *add* is $S_a = (P_1 \lhd R_1 \lhd P_1^{R_1} \wedge sentinel \lhd G_1 \lhd R_1^* | E_1 | R_1^* \lhd A_1)$ with $P_1 \overset{def}{=} Buf_{a1} = \emptyset$, $R_1 \overset{def}{=} Count = \overset{\square}{Count}$, $G_1 \overset{def}{=} Buf = \overset{\square}{Buf} + e \triangleright elt$, $E_1 \overset{def}{=} G_1$, and $A_1 = \{\ Buf_{a1} = \{e \triangleright elt\} \prec (1 \lhd e \triangleright elt)\}$. The specification of the body related to S_a can be deduced using the await-rule.

On the other hand we customize the specification of *incr* to meet the requirements of the composition rule. We now have $S_i \overset{def}{=} (P_2 \lhd R_2 \lhd W_2 \lhd G_2 \lhd E_2 \lhd \emptyset)$ where $P_2 \overset{def}{=} true$, $R_2 \overset{def}{=} G_1$, $W_2 = false$, $G_2 \overset{def}{=} R_1$, $E_2 \overset{def}{=} Count = \overset{\square}{Count} + 1$. It can easily be verified that starting from $\mathcal{S}_0$, the premises of the composition rule are satisfied. We now subscribe *incr* to events with identifiers equal to 1. The composition

rule yields the following result: $S_a = (P_1 \frown R_1 \frown P_1^{R_1} \wedge sentinel \frown G_1 \vee G_2 \frown E_1 \wedge E_2 \frown \{\ Buf_{a1} = \{e \triangleright elt\} \prec (1 \frown e \triangleright elt)\})$.
This is indeed the specification of a program which adds an element in the buffer and increments the counter. The program blocks until *sentinel* becomes true. Further, executing programs will not be interrupted by the environment. Other programs will wait until *sentinel* becomes true and no other program is running. This means that announced events are not simply discarded as in the previous example.

7 Conclusion

"Concurrent programming is hard and shared variable programming is very hard [30]." Concurrent programming with synchronization, shared variable and event based communication is therefore "three times" harder. We presented some shortcomings of the existing approaches in this paper.

In addition, we proposed a logic (LECAP) that allows specifying applications based on these paradigms (concurrency, shared variables, events, synchronization). The logic also supports composition of these specifications into specifications of larger applications. LECAP is therefore intrinsically oriented towards construction of complex systems. The paper also gave the formal semantics of the LECAP programming language. Such a language is based on an event based system whose formal definition was presented. Jones's rely/guarantee approach for the construction of interfering programs was extended with announcement conditions. The paper finally presented two examples that illustrate the approach.

It is obvious that a software development method cannot be established based on two examples. We therefore need to develop more examples and case studies to further experiment our approach. Additionally, refinement and verification of LECAP specifications are tasks we have to tackle.

Acknowledgments. We would like to acknowledge the helpful comments of Clemens Kerer, Gerald Reif, and Joe Oberleitner. We have also gratefully appreciated the useful comments received from Ketil Stølen.

References

1. P. Aczel. An inference rule for parallel composition, University of Manchester, 1983.
2. Stern Agerholm and Peter Gorm Larsen. A Lightweight Approach to Formal Methods. In *Proceedings of the Interna-tional Workshop on Currents Trends in Applied Formal Methods*. Springer Verlag, October 1998.
3. Apple Computer. *Inside Macintosh*, volume 6. Addison Wesley, 1991.
4. K.P. Birman. The progress group approach to reliable distributed computing. *Communications of the ACM*, 12:37–53, December 1993.
5. K. Brockschmidt. *Inside OLE*. Microsoft Press, Redmond, 1995.
6. C.A.R Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

7. A. Carzaniga, D.S. Rosenblum, and A.L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 3(19):332–383, August 2001.
8. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison Wesley, 1988.
9. G. Cugola, E. Di Nitto, and A. Fuggetta. The JEDI Event-Based Infrastructure and its Application to the Development of the OPSS WFMS. *Transaction of Software Engineering (TSE)*, 27(9), September 2001.
10. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
11. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasonning about Implicit Invocation. In *Proceedings of the 6th International Symposium on the Foundations of Software Engineering, FSE-6*. ACM, 1998.
12. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Towards a formal treatment of implicit invocation using rely/guarantee reasoning. *Formal Aspects of Computing*, 10, 1998.
13. C. Ene and Traian Muntean. A broadcast-based calculus for communicating systems. Technical report, Laboratoire d'Informatique de Marseille, 2000.
14. Object Management Group. OMG Formal Documentation. Technical report, OMG, December 1999.
15. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 1969.
16. D. Jackson and J. Wing. *Lightweight Formal Methods*. In IEEE Computer. Springer Verlag, April 1996.
17. C.B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and Systems*, 5(4), October 1983.
18. Cliff B. Jones. *Systematic software development using VDM*. Prentice-Hall International, 1990. 2nd edition.
19. B. Krishnamurthy and N.S. Barghouti. Provence: A process visualization and enactment environment. In *Proceedings of 4th European Software Engineering Conference*, pages 451–465, 1993.
20. R. Milner. *The Calculus of Communicating Systems*. Prentice Hall, 1993.
21. Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Client/Server Survival Guide*. Wiley Computer Publishing, second edition, 1996.
22. S. Owicki and D. Gries. Verifying properties of parallel programs: an axiomatic approach. *Communications of the ACM*, 19(5), May 1976.
23. K. Prasad. A calculus of broadcasting systems. In *Proceedings of TAPSOFT'91*, volume 493, 1991.
24. J.M. Purtilo. The polylith software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.
25. S.P. Reiss. Connecting tools using message passing in the field program development environment. *IEEE Software*, 19(5), July 1990.
26. Ed Roman and Scott W. Ambler and Tyler Jewell. *Mastering Enterprise JavaBeans*. John Wiley & Sons, second edition, 2002.
27. K. Stolen. A Method for the Development of Totally Correct Shared-State Parallel Programs. In *CONCUR'91*, pages 510–525. Springer Verlag, 1991.
28. SunSoft. *The ToolTalk Service: An Inter-operability Solution*. Prentice-Hall, 1993.
29. Peter Sutton, Rhys Arkins, and Bill Segall. Supporting disconnectedness-transparent information delivery for mobile and invisible computing. In *Proceedings of 2001 IEEE International symposium on Cluster Computing and the Grid (CCGrid'01)*, May 2001.

30. Jim Woodcock and Arthur Hughes. Unifying theories of parallel programming. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM 2002)*. Springer Verlag, 2002.
31. Q. Xu, W.-P. de Roever, and J. He. The rely guarantee method for verifying shared variable concurrent programs. *Formal Aspects of Computing*, 9:149–174, 1997.
32. Q. Xu and J. He. A theory of state-based parallel programming by refinement: part 1. In *Proceedings of the 4th BCS-FACS Refinement Workshop*. Springer Verlag, 1991.

A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems

Stephan Merz[1], Martin Wirsing[2], and Júlia Zappe[2]

[1] INRIA Lorraine, LORIA, Nancy
Stephan.Merz@loria.fr
[2] Institut für Informatik, Ludwig-Maximilians-Universität München
{wirsing,zappe}@informatik.uni-muenchen.de

Abstract. We define a variant of Lamport's Temporal Logic of Actions, extended by spatial modalities, that is intended for the specification of mobile systems with distributed state. We discuss notions of refinement appropriate for mobile systems, specifically concerning the topological structure of the system, and show how these can be represented in the logic via quantification and implication, ensuring transitivity and compositionality of refinements.

1 Background

The design of systems that make use of mobile code has recently found wide attention. Advances in network technology sometimes make it more attractive to transmit code for execution at remote sites than to rely on more conventional architectures such as client-server systems. It has quickly become apparent that the design of mobile systems requires specific abstractions that should be reflected in formal development methods and in underlying calculi and logics.

Milner's calculus [9] has started a line of research that investigates foundational calculi for mobile systems, e.g. [4,5,10,13], emphasizing different aspects of mobility and offering different primitives to describe the interaction of mobile components. In particular, the Ambient Calculus due to Cardelli and Gordon introduced the notion of (nested and dynamically reconfigurable) named administrative domains that must be crossed by mobile code.

Some of these calculi have been complemented by logics that allow to express run-time properties of mobile systems [2,3,11]. These logics typically include both spatial and temporal modalities to reflect the topological system structure as well as its evolution over time. Formulas of these logics are evaluated over process terms via an *intensional* semantics [12] and closely reflect the syntactic structure of processes. This close correspondence makes it difficult, if not impossible, to refine the process terms during system development while preserving the formulas of the logic. In this regard, such logics are inadequate for use as *specification logics* that could underly a method of stepwise system development based on refinement.

In the present paper we follow a different approach and define a spatio-temporal specification logic whose semantics is based on a notion of system behaviors similar to standard (linear-time) temporal logics, and independent of any specific operational

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 87–101, 2003.

calculus. Our main goal in the design of the logic is to support refinement: properties established at a higher level of abstraction should be preserved in the implementation. In the context of reactive systems, this goal has successfully been achieved in Lamport's Temporal Logic of Actions [7]. We therefore follow the general philosophy of TLA, but add spatial modalities to express the topology of configurations. We also discuss, informally, concepts of refinement that arise in the development of mobile systems, and how to represent those in the logic. Besides classical temporal (or operation) refinement that is already supported in TLA since its formulas are invariant under stuttering equivalence, we identify two concepts that are more specific to mobile systems, namely *spatial extension* and *virtualisation of locations*. We show that these concepts can also be represented in our logic via implication and novel forms of quantification that express hiding of local state and of agent names, respectively.

The outline of the paper is as follows: Section 2 introduces Mobile TLA (MTLA) at the hand of a simple agent system. Section 3 gives a more formal account of the syntax and semantics of simple MTLA, the logic that we use to specify closed mobile systems. Section 4 studies refinement principles for mobile systems that motivate extensions of the logic by two forms of quantification. Finally, Sect. 5 summarizes our contributions and indicates future work.

2 Example: Joe's Shopping Agent

The specification of a mobile system describes relevant aspects of network topology as well as of the dynamic system behavior, including the movement and interaction of agents. We identify all network locations by unique (or physical) names. In informal discussions, we do not distinguish between names and the domains or agents they represent. Domains may be nested, giving rise to a tree structure of names, and mobility is formally represented by structural changes of these trees that result from agents moving across domain boundaries.

As our running example, we consider the specification of a simple shopping agent that scans a network in the search of the best offer for some item. We assume a finite, fixed set Net of names that represent (immobile) network locations that the agent may visit during its search; $joe \in Net$ denotes the agent's home location. The name $shopper \Subset Net$ is used to denote the mobile shopping agent itself. Its local state is described by three state variables: the variable ctl indicates the control state of the shopping agent; it may assume the values "idle" and "shopping". When state equals "shopping", the variable $item$ indicates the good the shopper is searching for, and $found$ holds the set of offers that the agent has collected so far. For the locations $n \in Net$ we assume the state variable $offers$ to represent the catalogue of goods they offer, and the state variable id to denote a (logical) network name that is used by the agent to remember the origin of an offer.

A high-level MTLA specification appears in Fig. 1 as formula $Shopper$.[1] We now informally explain its meaning; the formal definition of MTLA is postponed to Sect. 3.

[1] We adopt Lamport's convention [6] of writing multi-line conjunctions and disjunctions as lists whose items are labelled with the respective connective, using indentation to suppress parentheses.

$$
\begin{aligned}
\mathit{Init} \;\equiv\;& \mathit{joe}.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper}.\mathit{ctl} = \text{"idle"} \\
\mathit{Prepare}(x) \;\equiv\;& \wedge\, \mathit{shopper}\langle\mathbf{true}\rangle \wedge \circ\mathit{shopper}\langle\mathbf{true}\rangle \\
& \wedge\, \mathit{shopper}.\mathit{ctl} = \text{"idle"} \\
& \wedge\, \mathit{shopper}.\mathit{item}' = x \wedge \mathit{shopper}.\mathit{found}' = \emptyset \\
& \wedge\, \mathit{shopper}.\mathit{ctl}' = \text{"shopping"} \\
\mathit{GetOffer} \;\equiv\;& \wedge\, \mathit{shopper}\langle\mathbf{true}\rangle \wedge \circ\mathit{shopper}\langle\mathbf{true}\rangle \\
& \wedge\, \mathit{shopper}.\mathit{ctl} = \text{"shopping"} \wedge \mathit{shopper}.\mathit{item} \in \mathit{offers} \\
& \wedge\, \mathit{shopper}.\mathit{found}' = \mathit{shopper}.\mathit{found} \\
& \qquad\qquad \oplus \{(\mathit{id}, \mathit{getOffer}(\mathit{offers}, \mathit{shopper}.\mathit{item}))\} \\
& \wedge\, \textsc{unchanged}(\mathit{shopper}.\mathit{ctl}, \mathit{shopper}.\mathit{item}) \\
\mathit{PickOffer} \;\equiv\;& \wedge\, \mathit{shopper}\langle\mathbf{true}\rangle \wedge \circ\mathit{shopper}\langle\mathbf{true}\rangle \\
& \wedge\, \mathit{shopper}.\mathit{ctl} = \text{"shopping"} \wedge |\mathit{shopper}.\mathit{found}| \geq 3 \\
& \wedge\, \mathit{bestOffer}' = \mathit{pickOffer}(\mathit{shopper}.\mathit{found}, \mathit{shopper}.\mathit{item}) \\
& \wedge\, \mathit{shopper}.\mathit{ctl}' = \text{"idle"} \\
\mathit{Move}_{n,m} \;\equiv\;& \wedge\, n.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper}.\mathit{ctl} = \text{"shopping"} \\
& \wedge\, n.\mathit{shopper} \gg m.\mathit{shopper} \\
\mathit{vars} \;\equiv\;& \mathit{shopper}.\mathit{ctl}, \mathit{shopper}.\mathit{item}, \mathit{shopper}.\mathit{found} \\
\mathit{JoeActions} \;\equiv\;& (\exists x : \mathit{Prepare}(x)) \vee \mathit{PickOffer} \\
\mathit{Shopper} \;\equiv\;& \wedge\, \mathit{Init} \\
& \wedge\, \Box \textstyle\bigwedge_{n,m \in \mathit{Net}} n\langle m[\mathbf{false}]\rangle \\
& \wedge\, \Box\big[\mathit{joe}[\mathit{JoeActions}] \vee \textstyle\bigvee_{n \in \mathit{Net}} n[\mathit{GetOffer}]\big]_{\mathit{vars}} \\
& \wedge\, \textstyle\bigwedge_{n \in \mathit{Net}} \Box\big[\bigvee_{m \in \mathit{Net}} \mathit{Move}_{n,m}\big]_{-n.\mathit{shopper}}
\end{aligned}
$$

Fig. 1. Specification of a simple shopping agent

The first conjunct Init of specification $\mathit{Shopper}$ asserts the specifications's initial condition. It requires the shopping agent to be at its home location and its control state to be "idle". We write $n[F]$ to assert that F holds at location n, provided n exists. The formula $n\langle F\rangle$ also requires F to hold at n, but moreover asserts the existence of location n.

The second conjunct of $\mathit{Shopper}$ describes part of the network topology: it requires the (immobile) locations $n \in \mathit{Net}$ to be present at all instances of time, and not to be nested inside one another.

The third conjunct of formula $\mathit{Shopper}$ specifies the allowed changes to the shopping agent's local state. Similarly as in TLA, a formula $\Box[A]_v$ requires every transition that changes v to satisfy the transition formula A, which may refer to the post-state of the transition via either the next-state operator $\circ$ or primed state variables. Our example allows three kinds of transitions: at location joe, one of the $\mathit{Prepare}$ or $\mathit{PickOffer}$ actions may occur, whereas transitions described by $\mathit{GetOffer}$ are allowed to occur at any location $n \in \mathit{Net}$ (even including the shopper's home location). For example, $\mathit{GetOffer}$ requires the shopper to be and remain at the location of evaluation, in state "shopping", and the target item to be among the goods offered at the current location. The agent then inserts a pair $(\mathit{id}, \mathit{val})$ into the set found of collected offers where id is the logical identity of the current location and val is the offer as denoted by the (static) function $\mathit{getOffer}$. The other transition specifications can be interpreted similarly.

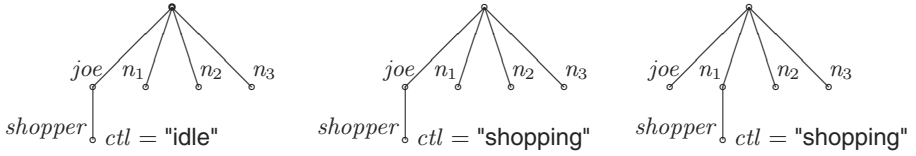


Fig. 2. Prefix of a run

The final conjunct of specification $Shopper$ describes the shopper's movements about the network. A formula $\Box[A]_{-n}$ asserts that every transition such that $n\langle\mathbf{true}\rangle$ is true before and false after the transition has to satisfy A. In our example, we require action $Move_{n \leftarrow m}$ to be performed, for some location $m \in Net$, whenever the shopper leaves domain n. The conjunct $n \triangleright\!\!\triangleright shopper \gg m \triangleright\!\!\triangleright shopper$ that appears in the description of $Move_{n \leftarrow m}$ requires the subtree denoted by $shopper$ located below the domain n to move to the domain m without changing the agent's local state.

A more complete specification of the shopping agent would also include fairness and liveness properties which we have omitted because they do not play an important role for the remainder of this paper.

Besides system specifications, MTLA formulas can also express system properties, including invariants. For example, the shopping agent described by the specification in Fig. 1 is always located beneath some network node, and therefore the implication

$$Shopper \Rightarrow \Box \bigvee_{n \in Net} n \triangleright\!\!\triangleright shopper\langle\mathbf{true}\rangle$$

is valid. Similarly, the implication

$$Shopper \Rightarrow \Box(shopper \triangleright\!\!\triangleright ctl = \text{"idle"} \Rightarrow joe \triangleright\!\!\triangleright shopper\langle\mathbf{true}\rangle)$$

asserts that the shopping agent can be in its idle state only if it is at its home location.

3 Simple Mobile TLA

We describe the topological structure of a mobile system at any given instant as a finite tree t whose edges are labelled with unique ("physical") names n drawn from a denumerably infinite set N, as depicted in Fig. 2. Equivalently, instead of labelling edges we often consider the labels to be attached to the target nodes, such that every node of t except for the root carries a unique label. Moreover, with every node of the tree we associate a local state.

Technically, a *configuration* is a pair $(t \leftarrow \square)$: the tree t is given by a prefix-closed set $t \subseteq \mathsf{N}^*$ such that for any $\square\, n \leftarrow \square\, m \in t$ we have $n = m$ only if $\square = \square$; the empty word $\square$ denotes the root of the tree. For every node $\square \in t$, the local state $\square(\square)$ is a valuation of the state variables as explained below. A *run* of a system is represented as an $\square$-sequence $\square = (t_0 \leftarrow \square_0)(t_1 \leftarrow \square_1) \triangleright\!\!\triangleright\!\!\triangleright$ of configurations, cf. Fig. 2. For technical reasons we require that $t_i \neq \emptyset$ for all $i \in \mathbb{N}$. Transitions may change the local state at some nodes, but also modify the tree structure; structural changes represent the movement of

agents across administrative domains or the creation or destruction of agents. For a run $\sigma = (t_0, \sigma_0)(t_1, \sigma_1) \triangleright\triangleright\triangleright$ and $i \in \mathbb{N}$, we denote by $\sigma|_i$ the suffix $(t_i, \sigma_i)(t_{i+1}, \sigma_{i+1}), \triangleright\triangleright\triangleright$.

A tree t induces a partial order on the names that it contains. More precisely, we write $m \preceq_t n$ for $m, n \in \mathsf{N}$ and say that m *is below* n in t iff $\sigma n \tau \in t$ for some $\sigma, \tau \in \mathsf{N}^*$ such that $\sigma n \tau$ ends in m. For a tree t and a name $n \in \mathsf{N}$, we write

$$t{\downarrow}n = \{\sigma \in \mathsf{N}^* \mid \sigma n \tau \in t \text{ for some } \tau \in \mathsf{N}^*\}$$

for the subtree of t rooted at the (unique) node labelled by n. If no such node exists, $t{\downarrow}n$ denotes the empty tree $\emptyset$. This notation is extended to sequences $\sigma \in \mathsf{N}^*$ by defining $t{\downarrow}\varepsilon = t$ and $t{\downarrow}\sigma n = (t{\downarrow}\sigma){\downarrow}n$. For a name n that occurs in a tree t, i.e. such that $t{\downarrow}n \neq \emptyset$, we write $\sigma(t, n)$ for the unique path $\sigma n \in t$ ending in name n. By N^+ we denote the set $\mathsf{N} \cup \{\sigma\}$ (we assume that $\sigma \notin \mathsf{N}$). We extend some of our notation to N^+ by letting $\sigma(t, \sigma) = \sigma$ if $t \neq \emptyset$ and defining $n \preceq_t \sigma$ (where $n \in \mathsf{N}^+$) iff $t{\downarrow}n \neq \emptyset$, and $\sigma \preceq_t n$ to hold for no $n \in \mathsf{N}$. We write $m \prec_t n$ if $m \preceq_t n$ and $m \neq n$.

The connectives of MTLA extend classical first-order logic by spatial and temporal modalities. We also add an operator to describe structural modifications of trees during transitions. Formally, we define (pure) formulas and terms as well as "impure" ones; the latter generalize the transition formulas of TLA [7]. We assume given a signature (consisting of function and predicate symbols) of first-order logic with equality and denumerable sets $\mathcal{V}_r$ and $\mathcal{V}_f$ of rigid and flexible individual variables. The semantics of MTLA assumes a first-order interpretation $\mathcal{I}$ (defining a non-empty universe $|\mathcal{I}|$ and interpretations of the function and predicate symbols). Terms and formulas are interpreted with respect to a run whose valuations interpret the flexible variables, an index $n \in \mathsf{N}^+$ that indicates the "location of evaluation", and a valuation of the rigid variables. In the following inductive definition, clauses 1–8 as well as 11, 12, and 14 are standard [7,8] whereas clauses 9, 10, 13, and 15 introduce the spatial extensions of MTLA.

Definition 1. *Assume a fixed first-order interpretation $\mathcal{I}$, a run $\sigma = (t_0, \sigma_0)(t_1, \sigma_1) \triangleright\triangleright\triangleright$ with $\sigma_i : t_i \to (\mathcal{V}_f \to |\mathcal{I}|)$, and a valuation $\sigma : \mathcal{V}_r \to |\mathcal{I}|$. We define the terms and formulas of MTLA and their semantics, for arbitrary $n \in \mathsf{N}^+$.*

1. *Every variable $x \in \mathcal{V}_r \cup \mathcal{V}_f$ is a (pure) term. For a rigid variable $x \in \mathcal{V}_r$, we define $\sigma^{(n,\sigma)}(x) = \sigma(x)$. For a state variable $x \in \mathcal{V}_f$, we let $\sigma^{(n,\sigma)}(x) = \sigma_0(\sigma(t_0, n))(x)$ be the value assigned to x by the local interpretation associated with node n of the initial configuration in σ, provided that $t_0{\downarrow}n \neq \emptyset$; otherwise $\sigma^{(n,\sigma)}(x)$ is an arbitrary but fixed element $a \in |\mathcal{I}|$.*
2. *If $t_1, \triangleright\triangleright\triangleright, t_k$ are (im)pure terms and f is a k-ary function symbol then $f(t_1, \triangleright\triangleright\triangleright, t_k)$ is again an (im)pure term whose interpretation is given by $\sigma^{(n,\sigma)}(f(t_1, \triangleright\triangleright\triangleright, t_k)) = \mathcal{I}(f)(\sigma^{(n,\sigma)}(t_1), \triangleright\triangleright\triangleright, \sigma^{(n,\sigma)}(t_k))$.*
3. *If A is an (im)pure formula and $x \in \mathcal{V}_r$ then $\sigma x : A$ is an (im)pure term where $\sigma^{(n,\sigma)}(\sigma x : A)$ is some value $a \in |\mathcal{I}|$ such that $\sigma, n, \sigma[x := a] \models A$ if some such value exists, otherwise $\sigma^{(n,\sigma)}(\sigma x : A)$ is some arbitrary but fixed value $a \in |\mathcal{I}|$.*
4. *Every pure term or formula is also an impure term or formula.*
5. *If P is a k-ary predicate symbol and $t_1, \triangleright\triangleright\triangleright, t_k$ are (im)pure terms then $P(t_1, \triangleright\triangleright\triangleright, t_k)$ is an (im)pure formula. We define $\sigma, n, \sigma \models P(t_1, \triangleright\triangleright\triangleright, t_k)$ and say that $P(t_1, \triangleright\triangleright\triangleright, t_k)$ holds at location n in σ, iff $(\sigma^{(n,\sigma)}(t_1), \triangleright\triangleright\triangleright, \sigma^{(n,\sigma)}(t_k)) \in \mathcal{I}(P)$.*

6. **false** *is a pure formula that holds nowhere:* $\sigma, n, \rho \not\models \mathbf{false}$.
7. *If* A, B *are (im)pure formulas then so is* $A \Rightarrow B$. *We define* $\sigma, n, \rho \models A \Rightarrow B$ *iff* $\sigma, n, \rho \not\models A$ *or* $\sigma, n, \rho \models B$.
8. *If* A *is an (im)pure formula and* $x \in \mathcal{V}_r$ *then* $\exists x : A$ *is again an (im)pure formula, and* $\sigma, n, \rho \models \exists x : A$ *iff* $\sigma, n, \rho[x := a] \models A$ *for some* $a \in |\mathcal{I}|$.
9. *If* A *is an (im)pure formula and* $m \in \mathsf{N}$ *then* $m[A]$ *is again an (im)pure formula whose interpretation is given by* $\sigma, n, \rho \models m[A]$ *iff* $m \prec_{t_0} n$ *implies* $\sigma, m, \rho \models A$.
10. *If* A *is an (im)pure formula then* $\overline{\Box} A$ *("everywhere* A*") is again an (im)pure formula, and* $\sigma, n, \rho \models \overline{\Box} A$ *iff* $\sigma, m, \rho \models A$ *for all* $m \in \mathsf{N}^+$ *such that* $m \preceq_{t_0} n$.
11. *If* F *is a pure formula then* $\Box F$ *("always* F*") is a pure formula with semantics* $\sigma, n, \rho \models \Box F$ *iff for all* $i \in \mathbb{N}$, $t_j{\downarrow}n = \emptyset$ *for some* $j \leq i$ *or* $\sigma|_i, n, \rho \models F$.
12. *If* F *is a pure formula then* $\circ F$ *("next-time* F*") is an impure formula, and we define* $\sigma, n, \rho \models \circ F$ *iff* $t_1{\downarrow}n = \emptyset$ *or* $\sigma|_1, n, \rho \models F$.
13. *For* $m \in \mathsf{N}$ *and* $\sigma, \rho \in \mathsf{N}^*$, $\sigma \rhd m \gg \rho \rhd m$ *is an impure formula whose semantics is defined by* $\sigma, n, \rho \models \sigma \rhd m \gg \rho \rhd m$ *iff both* $t_0{\downarrow}n \cdot m = t_1{\downarrow}n \cdot m$ *and* $\sigma_0(\sigma(t_0 \cdot m)\sigma) = \sigma_1(\sigma(t_1 \cdot m)\sigma)$ *for all* $\sigma \in t_0{\downarrow}n \cdot m$.
14. *If* A *is an impure formula and* t *is a pure term then* $\Box[A]_t$ *is a pure formula, and* $\sigma, n, \rho \models \Box[A]_t$ *iff for all* $i \in \mathbb{N}$, $t_j{\downarrow}n = \emptyset$ *for some* $j \leq i$ *or* $\sigma|_i^{(n,\sigma)}(t) = \sigma|_{i+1}^{(n,\sigma)}(t)$ *or* $\sigma|_i, n, \rho \models A$.
15. *If* A *is an impure formula and* S *is a non-temporal formula, i.e., built only using rules 1–10, then* $\Box[A]_S$ *is a pure formula with semantics* $\sigma, n, \rho \models \Box[A]_S$ *iff for all* $i \in \mathbb{N}$, $t_j{\downarrow}n = \emptyset$ *for some* $j \leq i$ *or* $\sigma|_i, n, \rho \models S$ *iff* $\sigma|_{i+1}, n, \rho \models S$ *or* $\sigma|_i, n, \rho \models A$.

We say that F *holds of* σ, *written* $\sigma, \rho \models F$ *iff* $\sigma, \sigma, \rho \models F$. *Formula* F *is* valid, *written* $\models F$, *iff* $\sigma, \rho \models F$ *holds for all runs* σ *and valuations* ρ.

Like TLA, MTLA is a linear-time temporal logic: formulas are interpreted over linear sequences of states. However, terms and formulas of MTLA are evaluated relative to a location, identified by a name n. Intuitively, the point of evaluation "follows" the movements of n in the tree. Because names may be created and deleted, the temporal operators of MTLA are effectively restricted to the possibly finite life-spans of a name, which ends when the name disappears from a tree. In particular, we consider a possible reappearance of a name in a later tree to represent a new domain that happens to reuse the same physical name.

The spatial modalities $m[\_]$ and $\overline{\Box}$ shift the spatial focus of evaluation. A formula $m[F]$ is "weak" in the sense that it holds trivially if the name m does not occur below the current point of evaluation. Moreover, note that the operator $m[\_]$ "looks arbitrarily far inside" the tree; as we will argue in Sect. 4.2, this is important if we want to refine a single domain by a hierarchy of domains. The "everywhere" operator refers to all nodes of the subtree rooted at the current point of evaluation.

The distinction between pure and impure formulas in the definition of a variant of TLA was introduced in [8] where it was shown that such a mutually recursive definition (compared to a tier of temporal formulas on top of a tier of transition formulas as in TLA) makes the logic more expressive while simplifying its axiomatization. "Impureness" is introduced by the next-time operator $\circ$ of linear-time temporal logic or by the "move"

operator $\gg$; impure formulas must be guarded by the $\Box[\_]_t$ or $\Box[\_]_S$ operators to produce a pure formula. This syntactic restriction ensures that all pure MTLA formulas are invariant under finite stuttering. Refinements are therefore allowed to introduce low-level steps that are invisible at the level of the original specification. Whereas formulas $\Box[A]_t$ specify the allowed changes of local states, formulas $\Box[A]_S$ are used to describe structural modifications of trees.

The formula $\square \triangleright m \gg \square \triangleright m$ describes the movement of an agent m, including all enclosed sub-locations and their local states, from subdomain $\square$ to subdomain $\square$, which would be impossible to specify using just formulas of the form $n[F]$ that refer only to single locations, not to entire subtrees. If the path $\square\, m$ does not exist below the current location, the definition of $\square \triangleright m \gg \square \triangleright m$ requires m not to occur below $\square$ in the subtree after the transition.

When writing MTLA specifications we use many derived operators, beyond the standard abbreviations **true**, $\wedge$, $\vee$, and $\forall$. For a pure term t, we define the impure term $t' \equiv \square x : \bigcirc(t = x)$ to denote the value of t at the next instant. Similarly, for an (im)pure term t and a name $n \in \mathsf{N}$, $n \triangleright t$ denotes the (im)pure term $\square x : n[x = t]$ that denotes the value of t at sublocation n, provided such a location exists. For pure terms $t_1 \mathbin{‹} \triangleright\triangleright\triangleright \mathbin{‹} t_n$ we write $\textsc{unchanged}(t_1 \mathbin{‹} \triangleright\triangleright\triangleright \mathbin{‹} t_n)$ to denote the impure formula $t_1' = t_1 \wedge \triangleright\triangleright\triangleright \wedge t_n' = t_n$.

The formula $n\langle F\rangle \equiv \neg n[\neg F]$ is defined as the dual of $n[F]$; it requires the existence of a sublocation n such that F holds at n. To reduce the number of brackets, we sometimes write n (for a name $n \in \mathsf{N}$) instead of $n\langle\mathbf{true}\rangle$, asserting the existence of a location named n in the current tree, and write $n_1 \triangleright \cdots \triangleright n_k[F]$ and $n_1 \triangleright \cdots \triangleright n_k\langle F\rangle$ instead of $n_1[\cdots n_k[F]\cdots]$ and $n_1\langle\cdots n_k\langle F\rangle\cdots\rangle$.

The formula $\overline{\Diamond} P$ ("somewhere P") is defined as $\neg\overline{\Box}\neg P$ and holds of $\square$ if P holds at some sublocation. Similarly, $\Diamond F$ ("eventually P") is defined as the dual of $\Box F$; it requires F to hold eventually (within the life span of the current name). We write $\Diamond\langle A\rangle_t$ for $\neg\Box[\neg A]_t$, and similarly for $\Diamond\langle A\rangle_S$; these formulas hold if eventually t (resp., S) change value during a transition satisfying A. The formulas $\Box[A]_{-S}$ and $\Box[A]_{+S}$ abbreviate $\Box[S \Rightarrow A]_S$ and $\Box[\neg S \Rightarrow A]_S$; these formulas assert that A holds whenever the spatial formula S becomes false (resp., true) during a transition. Finally, the formula $\Box[A]_{u_1 \mathbin{‹}\triangleright\triangleright\triangleright\mathbin{‹} u_n}$ (where the u_i may be pure terms or pure spatial formulas) denotes $\Box[A]_{u_1} \wedge \triangleright\triangleright\triangleright \wedge \Box[A]_{u_n}$; it holds of $\square$ provided every transition that changes some u_i satisfies A.

Although we will mostly argue semantically, we list a few axioms of MTLA. It is easy to see that implication distributes over all operators of "rectangular shape":

$$\models \boxtimes(A \Rightarrow B) \Rightarrow (\boxtimes A \Rightarrow \boxtimes B) \quad \text{for } \boxtimes \in \{n[\_] \mathbin{‹} \overline{\Box} \mathbin{‹} \Box \mathbin{‹} \Box[\_]_u\}$$

The "everywhere" operator quantifies over all paths, so we have

$$\models \overline{\Box} F \Rightarrow F \qquad \text{and} \qquad \models \overline{\Box} F \Rightarrow n[F] \text{ for all } n \in \mathsf{N}$$

Finally, we have axioms that correspond to the assumed uniqueness of names and that express a form of "absorption" for the modalities $n[\_]$, which look arbitrarily deep into the tree. More precisely,

$$\models n[F] \Leftrightarrow \overline{\Box} n[F] \qquad \text{and} \qquad \models m \triangleright n\langle\mathbf{true}\rangle \Rightarrow (m \triangleright n[F] \Leftrightarrow n[F])$$

4 Refinement of Mobile Systems

The general idea behind refinement concepts is to allow a high-level description of a system to be replaced by a lower-level implementation while preserving the properties established at the higher level of abstraction. Concerning the refinement of mobile systems, we identify three basic principles of refinement that should be supported:

1. *Operation refinement* is a classical principle that is well-known from sequential and reactive systems. In particular, operations can be made more deterministic, and they can be decomposed into sequences of finer-grained actions.
2. *Spatial extension* can be used to decompose a single, high-level location n into a tree of sub-locations that collectively implement the behavior required of n, and whose root is again named n. In general, the local state originally associated with node n will be distributed among the locations of the implementation; it should then be hidden from the interface of the abstract specification.
3. *Virtualisation of locations* allows to replace a location of the abstract specification by a structurally different location hierarchy, with a different name. This form of refinement requires the name of the "virtualised" location to be hidden from the high-level interface.

A single refinement step may combine several of these principles. For example, we will see that a combination of operation refinement and virtualisation allows an atomic high-level move action to be implemented as a sequence of lower-level moves. We now consider each of the basic principles in more detail, motivating corresponding extensions of Simple MTLA, and illustrate them at the hand of our running example. Again, we follow the lead of TLA where refinement of a high-level specification Abs with internal variables aux by a low-level specification $Conc$ is expressed by validity of the implication

$$\models Conc \Rightarrow \exists\, aux : Abs$$

4.1 Operation Refinement

Figure 3 shows a specification of the shopping agent whose move actions have been restrained in two ways: first, moves from location n to another shop m are allowed only if the offers made at n (if any) have been entered in the agent's records and if m has not been visited before (i.e., its id does not appear in the agent's records). Second, moves to the home location are allowed only if the agent has recorded the offers made at n and if it has collected at least three offers. The restrained shopping agent's specification $RestrShopper$ is identical to formula $Shopper$ except for the fourth conjunct. It follows immediately from the definitions that both

$$MoveHome_n \Rightarrow Move_{n \cdot joe} \quad \text{and} \quad VisitShop_{n \cdot m} \Rightarrow Move_{n \cdot m}$$

are valid. Propositional logic and the monotonicity of the operator $\Box[\_]_S$ imply

$$\models RestrShopper \Rightarrow Shopper$$

$$
\begin{aligned}
\mathit{VisitShop}_{n,m} \;\equiv\; & \wedge\, n.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper.ctl} = \text{"shopping"} \\
& \wedge\, \mathit{shopper.item} \notin n.\mathit{offers} \vee n.\mathit{id} \in \mathrm{dom}(\mathit{shopper.found}) \\
& \wedge\, m.\mathit{id} \notin \mathrm{dom}(\mathit{shopper.found}) \\
& \wedge\, n.\mathit{shopper} \gg m.\mathit{shopper} \\
\mathit{MoveHome}_{n} \;\equiv\; & \wedge\, n.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper.ctl} = \text{"shopping"} \\
& \wedge\, \mathit{shopper.item} \notin n.\mathit{offers} \vee n.\mathit{id} \in \mathrm{dom}(\mathit{shopper.found}) \\
& \wedge\, |\mathit{shopper.found}| \geq 3 \\
& \wedge\, n.\mathit{shopper} \gg \mathit{joe.shopper} \\
\mathit{RestrShopper} \;\equiv\; & \wedge \ldots \\
& \wedge \textstyle\bigwedge_{n \in Net} \Box\big[\mathit{MoveHome}_{n} \vee \bigvee_{m \in Net \setminus \{Joe\}} \mathit{VisitShop}_{n,m}\big]_{-n.\mathit{shopper}}
\end{aligned}
$$

Fig. 3. Restraining the movement of the shopping agent

reflecting the fact that the specification of Fig. 3 is a possible refinement of the original specification shown in Fig. 1.

As in TLA, operation refinement based on the decomposition of a high-level action into sequences of implementation actions is also formally expressed by validity of implication. This is a consequence of the invariance of MTLA formulas under stuttering. We now turn to refinement principles that are specific to mobile systems because they change the topological structure of configurations.

4.2 Spatial Extension without Distribution of State

During system development, one may choose to implement a single location of the high-level specification by a hierarchy of locations. Semantically, this is reflected in a situation as illustrated in Fig. 4 where a single location n_1 is refined into a tree with root n_1 and new sub-locations in, out, and $dock$. Sublocations of the original location may be assigned to different sub-trees of the implementation. However, the spatial relations between the locations that are visible in the high-level specification are preserved.

In the context of the "shopping agent" example, let us assume that every network node is equipped with a designated "dock" location to hold visiting agents, and that incoming and outgoing agents are placed into "in" and "out" communication buffers. Figure 5 contains a specification of a corresponding version of the (original) shopping agent example. Formula $\mathit{DockedMove}_{n,m}$ describes a transition where the shopping

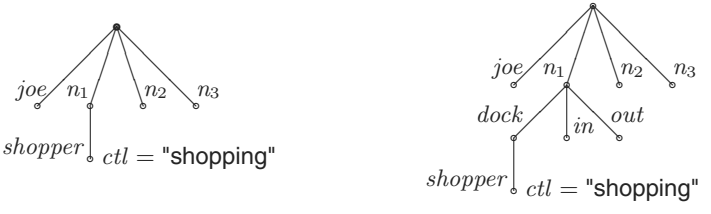


Fig. 4. Spatial extension of node n_1

$$
\begin{aligned}
\mathit{DockedInit} &\equiv \mathit{joe}.\mathit{dock}_{\mathit{joe}}.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper}.\mathit{ctl} = \text{"idle"} \\
\mathit{SendShopper}_n &\equiv \wedge\, n.\mathit{dock}_n.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper}.\mathit{ctl} = \text{"shopping"} \\
&\quad \wedge\, n.\mathit{dock}_n.\mathit{shopper} \gg n.\mathit{out}_n.\mathit{shopper} \\
\mathit{DockedMove}_{n,m} &\equiv \wedge\, n.\mathit{out}_n.\mathit{shopper}\langle\mathbf{true}\rangle \\
&\quad \wedge\, n.\mathit{out}_n.\mathit{shopper} \gg m.\mathit{in}_m.\mathit{shopper} \\
\mathit{RcvShopper}_n &\equiv \wedge\, n.\mathit{in}_n.\mathit{shopper}\langle\mathbf{true}\rangle \\
&\quad \wedge\, n.\mathit{in}_n.\mathit{shopper} \gg n.\mathit{dock}_n.\mathit{shopper} \\
\mathit{DockedShopper} &\equiv \wedge\, \mathit{DockedInit} \\
&\quad \wedge \ldots \\
&\quad \wedge \textstyle\bigwedge_{n \in \mathit{Net}} \wedge\, \Box[\mathit{SendShopper}_n]_{-\mathit{dock}_n.\mathit{shopper}} \\
&\qquad\qquad \wedge\, \Box[\mathit{RcvShopper}_n]_{-\mathit{in}_n.\mathit{shopper}} \\
&\qquad\qquad \wedge\, \Box\big[\textstyle\bigvee_{m \in \mathit{Net}} \mathit{DockedMove}_{n,m}\big]_{-\mathit{out}_n.\mathit{shopper}}
\end{aligned}
$$

Fig. 5. Network nodes with agent docks

agent is taken out of the out_n buffer associated with location n and migrates to the in_m buffer of location m. The formulas $\mathit{SendShopper}_n$ and $\mathit{RcvShopper}_n$ specify the movements between the in_n and out_n buffers and the dock_n location where the agent is hosted during its visit. (A more complete elaboration of this refinement would strengthen the preconditions of the other transition to assert that the agent is actually "in dock" and would describe any necessary packing, unpacking, and security checks before actually placing the agent in the dock.)

Specification $\mathit{DockedShopper}$ is a refinement of $\mathit{Shopper}$, and in fact, the implication

$$\mathit{DockedShopper} \Rightarrow \mathit{Shopper}$$

is valid. The proof relies on the invariant that the shopper can only be placed in the "out" buffer when it is in state "shopping", formally expressed by the invariant

$$\mathit{DockedShopper} \Rightarrow \bigwedge_{n \in \mathit{Net}} \Box(\mathit{out}_n \triangleright \mathit{shopper}\langle\mathbf{true}\rangle \Rightarrow \mathit{shopper} \triangleright \mathit{ctl} = \text{"shopping"})$$

which is easily seen to follow from the definition of formula $\mathit{SendShopper}_n$. Observe also that the moves between the dock and the communication buffers correspond to invisible, "stuttering" transitions of the original specification because the shopping agent stays within the respective network domain. Therefore, these actions are allowed by formula $\mathit{Shopper}$.

For spatial extension to be an admissible refinement principle, it is important that formulas $n[F]$ be interpreted as referring not just to a sublocation n immediately beneath the current location but to locations arbitrarily deep in the subtree.

4.3 Spatial Extension with Distribution of State

In the case of specification $\mathit{DockedShopper}$, we were able to represent spatial extension simply by implication because the "dock", "in", and "out" sub-locations did not introduce any local state. In general, spatial extension will be accompanied with a distribution

of the state variables associated with the high-level location n among the lower-level locations. Intuitively, such a distribution of local state is permissible provided that no other component attempts to directly access the local state at location n. In other words, state variables that are not part of the external "interface" of a specification may be distributed in the implementation. Because we do not wish to impose a fixed set of visibility rules, the specifier has to explicitly designate the interface in the specification by indicating which state variables should be hidden from the interface.

Logically, hiding local state components corresponds to existential quantification over flexible variables at certain locations. We therefore extend the syntax of MTLA formulas.

Definition 2. *The definition of MTLA formulas is extended by the following clause.*

16. *If F is a pure formula, $m \in \mathsf{N}$ is a name, and $v \in \mathcal{V}_f$ is a flexible variable then $\exists\!\!\exists\, m{\triangleright}v : F$ is again a pure formula.*

As in TLA [7], the semantics of quantification over flexible variables is somewhat complicated in order to preserve invariance under finite stuttering. We formally define *stuttering equivalence* as the smallest equivalence relation $\simeq$ on runs that identifies runs that differ by insertion or removal of duplicate configurations $(t_i \frown \sigma_i)$:

$$\triangleright\!\triangleright\!\triangleright(t_i \frown \sigma_i)(t_{i+1} \frown \sigma_{i+1})\triangleright\!\triangleright\!\triangleright \;\simeq\; \triangleright\!\triangleright\!\triangleright(t_i \frown \sigma_i)(t_i \frown \sigma_i)(t_{i+1} \frown \sigma_{i+1})\triangleright\!\triangleright\!\triangleright$$

It is straightforward to show that pure formulas of Simple MTLA are invariant w.r.t. stuttering equivalence, that is, for any MTLA formula F we have $\sigma \frown \sigma \models F$ iff $\sigma \frown \sigma \models F$ whenever $\sigma \simeq \sigma$.

We say that runs $\sigma = (s_0 \frown \sigma_0)(s_1 \frown \sigma_1)\triangleright\!\triangleright\!\triangleright$ and $\sigma = (t_0 \frown \theta_0)(t_1 \frown \theta_1)\triangleright\!\triangleright\!\triangleright$ are *equal up to $v \in \mathcal{V}_f$ at $m \in \mathsf{N}$*, written $\sigma =_{m\triangleright v} \sigma$ iff $s_i = t_i$ for all $i \in \mathbb{N}$ and $\sigma_i(\sigma)(x) = \theta_i(\sigma)(x)$ except possibly when $x \equiv v$ and $\sigma \equiv \sigma\triangleright m$ for some $\sigma \in \mathsf{N}^*$. In other words, the tree structures of the configurations in σ and σ have to be identical, and the local valuations may differ at most in the valuation assigned to variable v at nodes labelled m.

Finally, we define *similarity up to v at m* as the smallest equivalence relation $\approx_{m\triangleright v}$ that contains both $\simeq$ and $=_{m\triangleright v}$. We define the semantics of existential quantification over flexible variables by

$$\sigma \frown n \frown \sigma \models \exists\!\!\exists\, m{\triangleright}v : F \quad \text{iff} \quad \sigma \frown n \frown \sigma \models F \;\text{ for some } \sigma \approx_{m\triangleright v} \sigma$$

This definition clearly ensures that MTLA formulas of the form $\exists\!\!\exists\, m{\triangleright}v : F$ are again invariant w.r.t. stuttering equivalence.

As an example, we present another spatial extension of the shopping agent specification where we assume the databases containing the offers to reside in a sub-location db_n of each network node n. Its specification appears as formula $DBShopper$ in Fig. 6. The formula $GetOffer$ has been changed to access the database $offers$ hosted at the sub-location db_n instead of directly at location n. The new specification is a refinement of the original one when the variable $offers$ of each node $n \in Net$ is hidden from the interface; formally, the implication

$$DBShopper \Rightarrow \exists\!\!\exists\, n_1{\triangleright}offers \frown \triangleright\!\triangleright\!\triangleright \frown n_k{\triangleright}offers : Shopper$$

$$
\begin{aligned}
DBGetOffer_n \;\equiv\; & \wedge\, shopper\langle \mathbf{true}\rangle \wedge \circ shopper\langle \mathbf{true}\rangle \\
& \wedge\, shopper.ctl = \text{"shopping"} \wedge shopper.item \in db_n.offers \\
& \wedge\, shopper.found' = shopper.found \\
& \qquad\qquad \oplus \{(id \mapsto getOffer(db_n.offers, shopper.item))\} \\
& \wedge\, \textsc{unchanged}(shopper.ctl, shopper.item) \\
DBShopper \;\equiv\; & \wedge\, Init \\
& \wedge\, \Box \bigwedge_{m,n \in Net} n\langle db_n\langle \mathbf{true}\rangle \wedge m[\mathbf{false}]\rangle \\
& \wedge\, \Box\big[joe[JoeActions] \vee \bigvee_{n \in Net} n[DBGetOffer_n]\big]_{vars} \\
& \wedge\, \bigwedge_{n \in Net} \Box\big[\bigvee_{m \in Net} Move_{n,m}\big]_{-n.shopper}
\end{aligned}
$$

Fig. 6. Network nodes with sub-location hosting the database

is valid, assuming $Net = \{n_1 \triangleleft \cdots \triangleleft n_k\}$. Proofs of such refinements can be based on the following variant of the "refinement mapping" rule of TLA:

$$
F[t \triangleleft m \triangleright v] \Rightarrow \exists\, m \triangleright v : F \qquad \begin{array}{l}\text{provided all occurrences of } v \text{ in subformulas } \overline{\Box} G \text{ of } F \\ \text{are in the scope of some subformula } a[H] \text{ of } G\end{array}
$$

where t is a pure term and $F[t \triangleleft m \triangleright v]$ denotes the formula F where all top-level occurrences of v in any subformula $m[A]$ (i.e., those occurrences that are not in the scope of any further spatial operator) are replaced by t. For example, refinement of $Shopper$ by $DBShopper$ can be shown by the above rule by observing the validity of

$$
DBShopper \Rightarrow Shopper[db_{n_1} \triangleright offers \triangleleft n_1 \triangleright offers \triangleleft \cdots \triangleleft db_{n_k} \triangleright offers \triangleleft n_k \triangleright offers]
$$

4.4 Virtualisation of Locations

As the final and most general of our refinement principles for mobile systems, we consider the replacement of some location n in the abstract specification by a structurally different hierarchy of locations such that the externally observable behavior of the system is preserved. Unlike in the previous case of structural refinement with possible distribution of local state, this new case allows for implementations that do not contain an agent named n, implying that n itself, and not just its state components, should be hidden from the interface of the high-level specification. This motivates a second extension of the syntax of MTLA.

Definition 3. *The definition of MTLA is extended by quantification over names.*

17. If F is a pure formula and $m \in \mathsf{N}$ is a name then $\exists\, m : F$ is again a pure formula.

Intuitively, a run σ satisfies a formula $\exists\, m : F$ if at every configuration a subtree may be identified that "plays the role of m" as described by formula F. Formally, we require the existence of some sequence of configurations that differ from those in σ by *inserting* a new node that behaves as required at appropriate places of every configuration, and explicitly ensure closure under stuttering equivalence.

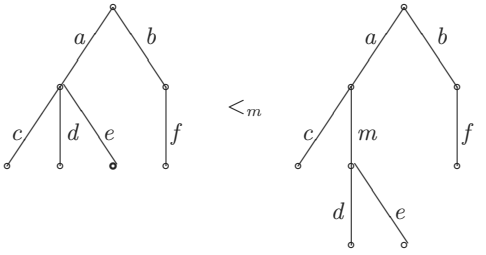


Fig. 7. Illustration of tree extension

For finite trees (with unique labellings) s and t and a name $m \in \mathsf{N}$ we define the relation $s <_m t$ to hold iff s results from t by removing the node labelled by m (if any); formally,

$$s <_m t \quad \text{iff} \quad \text{there exists } \square \in \mathsf{N}^* \text{ such that } s = \{\square \in t : \square m \,/\!\!\sqsubseteq \square\} \cup \{\square\square : \square m\square \in t\}$$

(see Fig. 7 for an illustration of this definition, choosing $\square = a$). The relation $<_m$ is extended to configurations in the canonical way by requiring that the local state associated with any node in s be that of the corresponding node in t, and arbitrary at the new node m:

$$(s \triangleleft \square) <_m (t \triangleleft \theta) \quad \text{iff} \quad s <_m t \quad \text{and} \quad \text{for all } \square \in s,\ \square(\square) = \begin{cases} \theta(\square) & \text{if } \square \in t \\ \theta(\square\, m\square) & \text{if } \square = \square\square \text{ where } \square m\square \in t \triangleleft \square \neq \square \end{cases}$$

Finally, the relation $<_m$ is extended to entire runs by

$$(s_0 \triangleleft \square_0)(s_1 \triangleleft \square_1) \rhd\!\rhd\!\rhd <_m (t_0 \triangleleft \theta_0)(t_1 \triangleleft \theta_1) \rhd\!\rhd\!\rhd \quad \text{iff} \quad (s_i \triangleleft \square_i) <_m (t_i \triangleleft \theta_i) \text{ for all } i \in \mathbb{N}.$$

The semantics of quantification over names is now defined by

$$\square \triangleleft n \triangleleft \square \models \exists\, m : F \quad \text{iff} \quad \text{there exist runs } \square \triangleleft \square \text{ such that } \square \simeq \square,\ \square <_l \square, \text{ and } \square \triangleleft n \triangleleft \square \models F[l \triangleleft m] \text{ for a name } l \text{ that does not occur in } \square \text{ or } F$$

We illustrate this refinement principle by an implementation that combines virtualisation and operation refinement to non-atomically move the shopping agent between network nodes via an intermediary location $transit \Subset Net$. (A subsequent application of spatial extension would allow that location to be refined into sub-locations to model movement across several network hops.)

Figure 8 contains the specification of a shopping agent that, starting at any location $n \in Net$, first moves to the intermediary location $transit$ before moving on to some other location $m \in Net$. Observe that the implication

$$SlowShopper \Rightarrow Shopper$$

is not valid because $SlowShopper$ does not satisfy the invariant that the shopping agent is always located at some location $n \in Net$. However, we do have

$$\models SlowShopper \Rightarrow \exists\, shopper : Shopper$$

$$
\begin{aligned}
\mathit{StartMove}_n &\equiv \wedge\, n.\mathit{shopper}\langle\mathbf{true}\rangle \wedge \mathit{shopper}.\mathit{ctl} = \text{"shopping"} \\
&\quad \wedge\, n.\mathit{shopper} \gg \mathit{transit}.\mathit{shopper} \\
\mathit{EndMove}_m &\equiv \wedge\, \mathit{transit}.\mathit{shopper}\langle\mathbf{true}\rangle \\
&\quad \wedge\, \mathit{transit}.\mathit{shopper} \gg m.\mathit{shopper} \\
\mathit{SlowShopper} &\equiv \wedge\, \mathit{Init} \\
&\quad \wedge\, \Box \bigwedge_{m,n \in \mathit{Net} \cup \{\mathit{transit}\}} n\langle m[\mathbf{false}]\rangle \\
&\quad \wedge\, \Box\big[\mathit{joe}[\mathit{JoeActions}] \vee \bigvee_{n \in \mathit{Net}} n[\mathit{GetOffer}]\big]_{\mathit{vars}} \\
&\quad \wedge\, \bigwedge_{n \in \mathit{Net}} \Box[\mathit{StartMove}_n]_{-n.\mathit{shopper}} \\
&\quad \wedge\, \Box\big[\bigvee_{m \in \mathit{Net}} \mathit{EndMove}_m\big]_{-\mathit{transit}.\mathit{shopper}}
\end{aligned}
$$

Fig. 8. Shopping agent with non-atomic moves

To see why that implication is valid, consider any run $\square$ of $\mathit{SlowShopper}$. We have to extend the configurations of $\square$ by a new location, say $\mathit{virtual}$, that indicates the current location of the $\mathit{shopper}$ agent of the original specification. Whenever the low-level $\mathit{shopper}$ agent is located at some node $n \in \mathit{Net}$, the same should be true of $\mathit{virtual}$. When $\mathit{shopper}$ is located at $\mathit{transit}$ in between transitions $\mathit{StartMove}_n$ and $\mathit{EndMove}_m$, the location $\mathit{virtual}$ should remain below location n, effectively delaying the high-level move action until the slow shopper arrives at its destination. At every configuration, the local state at location $\mathit{virtual}$ should be that of the slow shopper.

Proofs of refinements by virtualisation can be based on the rule

$$F[n \triangleleft m] \Rightarrow \exists\, m : F \quad \text{where } n \text{ does not occur in } F$$

However, this "refinement mapping" rule would have to be complemented by a rule for introducing "spatial history variables" [1] in order to prove that the specification $\mathit{Shopper}$ is refined by $\mathit{SlowShopper}$, since the location of the shopper prior to the last $\mathit{StartMove}$ transition has to be remembered in order to compute the location of the witness $\mathit{virtual}$.

Refinement by virtualisation allows more radical refinements than that of $\mathit{Shopper}$ by $\mathit{SlowShopper}$. For example, the formula $\mathit{Shopper}$, which employs a mobile agent, could be refined by a client-server solution that replaces mobility by communication. The proof idea would then be to place the virtual shopping agent at the node from which an offer is received, and to enforce additional stuttering transitions to simulate the Move actions. On the other hand, an implementation might use a swarm of agents instead of a single one.

5 Conclusion

We have defined an extension MTLA of Lamport's TLA by spatial modalities. The logic is intended for the specification of systems that exhibit mobility of agents across hierarchical domains. We have also considered different principles of refinement of mobile systems, focussing on refinements that change the spatial structure of the original specification. We have demonstrated that all these principles can be represented in MTLA by implication, possibly after appropriate hiding of state components or entire

agent hierarchies from the interface of the specification. In particular, transitivity and compositionality of refinement, expressed by the rules

$$\frac{S_1 \Rightarrow \exists\, x : S_2 \quad S_2 \Rightarrow \exists\, y : S_3}{S_1 \Rightarrow \exists\, x \cdot y : S_3} \qquad \frac{S_1 \Rightarrow \exists\, x : S_2}{S_1 \wedge S_3 \Rightarrow (\exists\, x : S_2) \wedge S_3}$$

are immediate consequences of standard propositional and quantifier rules that hold for MTLA. This should make MTLA a sound basis for the stepwise and compositional development of mobile systems.

More generally, we believe that "extensional" semantics such as ours provide a useful complement to existing "intensional" formalisms for mobile systems. Obviously, our definitions will have to be complemented by deductive verification rules to allow formal, syntactic verification of the properties and refinements that we have claimed of our examples. We also want to study the decidability of the model checking problem for our logic, applied to finite-state systems.

References

1. Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 81(2):253–284, May 1991.
2. Luis Caires and Luca Cardelli. A spatial logic for concurrency (part I). In *Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, pages 1–37. Springer-Verlag, 2001. Revised version to appear in Information and Computation.
3. Luca Cardelli and Andrew Gordon. Anytime, anywhere. Modal logics for mobile ambients. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 365–377. ACM Press, 2000.
4. Luca Cardelli and Andrew Gordon. Mobile ambients. *Theoretical Computer Science*, 240:177–213, 2000.
5. Cédric Fournet and Georges Gonthier. The reflexive chemical abstract machine and the Join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, St. Petersburg Beach, Florida, January 1996. ACM.
6. Leslie Lamport. How to write a long formula. Research Report 119, Digital Equipment Corporation, Systems Research Center, December 1993.
7. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
8. Stephan Merz. A more complete TLA. In J.M. Wing, J. Woodcock, and J. Davies, editors, *FM'99: World Congress on Formal Methods*, volume 1709 of *Lecture Notes in Computer Science*, pages 1226–1244, Toulouse, September 1999. Springer-Verlag.
9. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.
10. R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a kernel language for agents interaction and mobility. *IEEE Trans. on Software Engineering*, 24(5):315–330, 1998.
11. R. De Nicola and M. Loreti. A modal logic for Klaim. In T. Rus, editor, *Proc. Algebraic Methodology and Software Technology (AMAST 2000)*, volume 1816 of *Lecture Notes in Computer Science*, pages 339–354, Iowa, 2000. Springer-Verlag.
12. Davide Sangiorgi. Extensionality and intensionality of the ambient logic. In *Proc. of the 28th Intl. Conf. on Principles of Programming Languages (POPL'01)*, pages 4–17. ACM Press, 2001.
13. Jan Vitek and Giuseppe Castagna. Seal: A framework for secure mobile computations. In *ICCL Workshop: Internet Programming Languages*, pages 47–77, 1998.

Spatial Security Policies for Mobile Agents in a Sentient Computing Environment

David Scott[1], Alastair Beresford[1], and Alan Mycroft[2]

[1] Laboratory for Communications Engineering, University of Cambridge
[2] Computer Laboratory, University of Cambridge
William Gates Building, 15 JJ Thompson Avenue, Cambridge CB3 0FD, UK
{djs55,arb33}@eng.cam.ac.uk, am@cl.cam.ac.uk

Abstract. A Sentient Computing environment is one in which the system is able to perceive the state of the physical world and use this information to customise its behaviour. Mobile agents are a promising new programming methodology for building distributed applications with many advantages over traditional client-server designs. We believe that properly controlled mobile agents provide a good foundation on which to build Sentient applications.
The aims of this work are threefold: (*i*) to provide a simple location-based mechanism for the creation of security policies to control mobile agents; (*ii*) to simplify the task of producing applications for a pervasive computing environment through the constrained use of mobile agents; and (*iii*) to demonstrate the applicability of recent theoretical work using *ambients* to model mobility.

1 Introduction

The goal of pervasive computing is to create systems that *disappear* [18]—systems that fade into the background leaving users free to concentrate on their own tasks rather than explicitly "using the computer". Sentient Computing works towards this goal by adding *perception* to software; applications become more responsive and useful by observing and reacting to their physical environment [7]. The ability to sense the location of people and objects is an important building-block of such sentient systems. One of the most natural ways a program can react to user movement is to move itself around the network (e.g. consider a desktop "teleportation" program which moves a user's screen, mouse and keyboard to the computer nearest their current physical location). Mobile code opens up a number of tantalising possibilities, including: (*i*) exploiting resources near to the user's current location (e.g. multimedia hardware, keyboards and mice); (*ii*) supporting the illusion that user applications and their state is omnipresent—allowing a user to use any application from any location; and (*iii*) maximising efficiency by spreading resource-intensive tasks to where resources are under-used.

Unfortunately the use of mobile code has severe security implications [14]. Aiming to deploy mobile agent-based applications on the global Internet, the research community has concentrated on solving two difficult problems: (*i*) preventing malicious mobile code from gaining unauthorised access to resources controlled by the hosting machine; and (*ii*) stopping a malicious virtual machine learning secret information such as cryptographic keys by disassembling mobile agents. There is a third critical problem which is less

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 102–117, 2003.

well-studied: user control. How do we allow users to control the activities of mobile agents in an intuitive way, providing just enough security whilst still reaping the benefits of mobile code? How can users trust agents, written by other people, to respect their wishes when they enter their space and use their resources?

Clearly users need an understandable mechanism to constrain the behaviour of agents. They need a way to control what happens with things that matter to them in *their* world i.e. their data and their computers. Without this ability users will never fully trust mobile code technology and will never allow it to be used in practice.

To address this need, we propose a framework for creating spatial (i.e. location-based) security policies for mobile agents. This framework provides users with an easy to comprehend way to restrict and monitor the activities of agents within a Sentient Computing environment. By using location-based policies we hope to exploit structure which users are already familiar with. People are used to security policies governing physical spaces (e.g. "no unauthorised personnel are allowed in this area"); we extend this idea seamlessly into the ethereal world of mobile agents.

We believe that our framework will (*i*) promote the development of mobile-code based Sentient Computing applications; (*ii*) allow users fine-grained control over where agents are allowed to run, bypassing problems associated with situations where the agent does not trust the machine or vice-versa; and (*iii*) provide a demonstration of the applicability of recent theoretical work in the research community modelling mobility.

The remainder of this paper is structured as follows: Sect. 2 describes the design of our framework in detail. Examples of possible policies are found in Sect. 3. Related work may be found in Sect. 4, and Sect. 5 concludes.

2 Spatial Policy Framework

In this section we describe (*i*) how we model the world incorporating both physical objects and mobile agents; (*ii*) our language for expressing mobility security policies; and (*iii*) our proposal for an arbitration scheme to reconcile conflicting policies. Implementation details are omitted for brevity; they may be found elsewhere [15].

2.1 Overview

Users write security policies to influence both their own agents and any objects in which they have an interest (e.g. the computers in their office). Examples of policies might be "this agent should never leave my office" or "never let any agent enter this zone". The system continuously monitors the locations of both physical objects and mobile agents, keeping track of which policies are being violated. Since the system has no physical presence, it cannot act to block the movements of physical objects. Therefore the system cannot *guarantee* that policies will never be violated; instead policies are associated with an *action*, a command to be executed if and when a policy is violated (e.g. a command to kill the offending agent).

In contrast to physical objects, the system has full control over the life-cycle of mobile agents. Agents requesting permission to migrate between hosts will have their requests blocked if the movement would violate a policy. Agents which are *physically*

moved (typically by being carried on a laptop by a user) in violation of a policy may find themselves being suspended or killed.

Policies written by different people may conflict with each other. The system implements an arbitration scheme which exploits the natural structure of the spatial model to resolve these conflicts when they arise.

2.2 Modelling the World

We model the world as a tree of nested *entities*, analogous to *ambients* in the Ambient Calculus [2]. We begin our description by defining the following set of terms:

entity name: label used to name entities, equivalent to an entity minus any contents. Examples include the names of physical places, computers and mobile agents. By convention we use $\square$ to range over all entity names and a to range over mobile agent entity names.

entity: description of a particular location (given by an entity name) *along with* its contents. Note that, in a similar fashion to the Ambient Calculus, we are not restricted to describing only physical places but can represent any bounded region where activity happens. For example an office containing people may be described as an entity, as can a virtual machine containing mobile code. By convention we say that e ranges over entities.

path: sequence of entity names describing a route through the entity hierarchy naming a specific entity. Paths are written in the form $\square_1 \downarrow \rhd\rhd\rhd \downarrow \square_n$ and are described further in Sect. 2.4.

path expressions: regular expression-like facility to efficiently name a set of entities. Path expressions are described further in Sect. 2.4.

We divide our entity names into *sorts* each representing a different kind of object. The exact sorts used in any deployed system will depend on the kinds of things being modelled (e.g. an aviation-based system may introduce the sort "aircraft"). Here we restrict ourselves to use the following sorts:

room: a physical volume of space corresponding to buildings, offices etc.
person: an autonomous physical entity able to move between **room**s
workstation: an immovable physical object which can host computer processes
laptop: a mobile physical object which can host computer processes
context: a virtual machine capable of running mobile code
agent: a piece of mobile code

We write $e \lhd s$ to mean entity e is of sort s. The formula $SortContainable(s_1 \text{`} s_2)$ holds when entities of sort s_2 may be nested inside entities of sort s_1. This formula is defined graphically in Fig. 1. Intuitively, physical objects may nest in the obvious way (e.g. a **workstation** may nest inside a **room**). For convenience we define a relation $Containable(e_1 \text{`} e_2)$ which indicates that entity e_2 is permitted to nest inside e_1. These relations are related as follows:

$$Containable(e_1 \text{`} e_2) \Leftrightarrow e_1 \lhd s_1 \wedge e_2 \lhd s_2 \wedge SortContainable(s_1 \text{`} s_2)$$

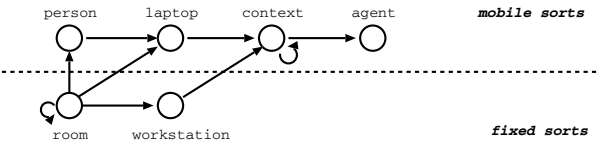


Fig. 1. The relation $SortContainable$

We define a partial function, $privs : entity \rightarrow permission\ set$, which is only defined on **context** entities and gives the set of permissions which are granted automatically to any **agent** nested inside. Examples of permissions include "can_play_sound" and "can_record_sound". This association between entities and sets of permissions allows us to use our spatial security policies to control more than simply the *location* of mobile agents; by creating appropriate **context**s we can control access to arbitrary resources. By convention, every computer (**workstation** or **laptop**) has at least one default **context** with an empty permission set.

A state of our world model may be written down with the following syntax (where $\square$ ranges over a set of entity names):

$$\begin{array}{lcll} entity & \leftarrow & entity \mid entity & \text{(siblings)} \\ entity & \leftarrow & \square[entity] & \text{(nesting in a place } \square\text{)} \\ entity & \leftarrow & \mathbf{0} & \text{(void)} \\ entity & \leftarrow & !\square & \text{(entity factory)} \end{array}$$

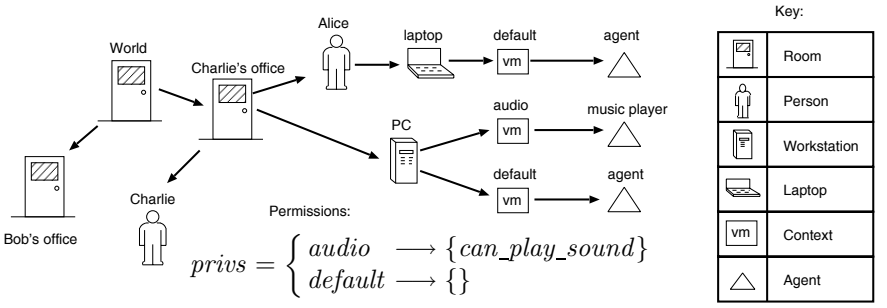
By convention we consider an entity factory $!\square$ to be a special kind of entity which can create other entities, i.e. the factory $!\square$ can spawn the entity $\square[\mathbf{0}]$, an empty entity ready for population. Note that every mobile agent which wishes to be created must be associated with at least one of these factory entities.

The entity $e = \square[e_1 \mid \bowtie \mid e_n]$ is well-sorted if $Containable(e \lhd e_1) \wedge \bowtie \wedge Containable(e \lhd e_n)$. The case $n = 0$ corresponds to the entity being empty i.e. $\square[\mathbf{0}]$. Observe that this syntax is similar to the subset of the ambient calculus which has no active processes and which describes only the structure of space, like that used in the semistructured data format described in [1].

As is conventional in mobility theory we next define a congruence relation, $\equiv$, under which entities are equal up to simple rearrangements of parts. In addition to reflexivity, symmetry, transitivity and context ($X \equiv Y \implies \square[X] \equiv \square[Y]$) this relation (often referred to as a structural congruence relation) is also commutative ($X \mid Y \equiv Y \mid X$), associative ($X \mid (Y \mid Z) \equiv (X \mid Y) \mid Z$) and "ignores zeros" ($X \mid \mathbf{0} \equiv X$).

2.3 Example

Consider a simple environment containing people, computers and several mobile agents. A graphical depiction of the model corresponding to this world at a particular time is displayed as follows:



There are various things to note about this configuration:

1. Alice is carrying a laptop inside Charlie's office. This laptop is currently running some mobile agent code. Note that such agents have potentially entered the room without having to migrate to a different host.
2. The PC in Charlie's office has been configured with an additional **context**, called `audio`. This **context** has been associated with a permission, *can play sound*, allowing agents to play sounds on a set of attached speakers. Therefore the **agent**, `music player` is able to play music in the office.

Although we have presented entity names as flat identifiers, they are likely to be more complicated in practice. For example they could contain secret data or be protected by a digital signature – possible benefits of such schemes include: (*i*) policies could be applied to whole classes of agents (e.g. all those signed by a particular key); or (*ii*) only people possessing the secret data would be able to successfully name an agent in a policy. For simplicity in the rest of this paper we will continue to use simple english names (like `music player`) for mobile agents.

2.4 Paths and Path Expressions

We uniquely name a single specific entity by providing a path from the root entity using the nesting relation, $\downarrow$. We say that $a \downarrow b$ if b is a child of a, i.e. b is contained within one level of nesting of a. A path to an entity will therefore have the form $\square_1 \downarrow \square_2 \downarrow \triangleright\triangleright\triangleright \downarrow \square_n$. For example, in the diagram in Sect. 2.3 an expression for the location of the entity `music player` would be `World` $\downarrow$ `Charlie's office` $\downarrow$ `PC` $\downarrow$ `audio` $\downarrow$ `music player`.

Path expressions, similar to regular expressions, are used to quantify over a set of paths. We first define the $\downarrow^*$ operator as the reflexive transitive closure of $\downarrow$ and then write the syntax of location expressions as follows:

$$
\begin{array}{rcll}
\mathit{element} & \leftarrow & \square & \text{(entity name)} \\
 & | & \{\square_1 \ldots \square_n\} & \text{(alternation)} \\
 & | & * & \text{(any)} \\
 & & & \\
\mathit{expression} & \leftarrow & \mathit{element} & \text{(root)} \\
 & | & \mathit{expression}/\mathit{element} & \text{(direct nesting)} \\
 & | & \mathit{expression}/\ldots/\mathit{element} & \text{(transitive nesting)}
\end{array}
$$

We define the *matching set* of a path expression exp as the set of paths $paths$ where $\forall p \in paths$ (with $p = p_1 \downarrow \rhd\rhd\rhd \downarrow p_n$)

- every step e_1/e_2 in exp corresponds with a step $p_1 \downarrow p_2$ in p where the element e_1 *matches* p_1 and e_2 *matches* p_2;
- every step $e_1/\ldots/e_2$ in exp corresponds to a sequence of steps $p_1 \downarrow \rhd\rhd\rhd \downarrow p_n$ for some n where the element e_1 *matches* p_1 and e_2 *matches* p_n;
- the element $\{\square_1, \square_2\}$ *matches* the entity with name $\square$ if $\square_1 = \square$ or $\square_2 = \square$;
- the element $*$ *matches* any entity name; and
- the trivial path element $\square$ *matches* an entity with name $\square$.

Path expressions provide a similar function to that of XPath [4], used for naming elements of XML documents.

2.5 Updating the Model

The model is updated dynamically to reflect the real-time configuration of the environment. Location sensors register changes in the physical configuration of the world (e.g. the movements of objects) which are then reflected by changes in the model. In addition, mobile agents may be programmatically created, frozen, killed or migrated, constrained only by the installed security policies. We define legal updates to the world by a labelled transition relation, $\xrightarrow{\square}$. We use labels to represent the side-effects of transitions, in particular the emission ($emit(\square)$) and reception ($receive(\square)$) of an agent during migration. The absence of a label on a transition indicates the lack of side-effects. A valid transition must have no labels at the top level – labels must always be matched and cancelled by the rule (migrate) described below. For brevity we write $a \leftrightarrow b$ if the transition is reversible i.e. if both $a \rightarrow b$ and $b \rightarrow a$ are legal transitions. The runtime system (described in a companion paper [15]) ensures that every event that occurs is represented by a legal transition.

For entities X, Y, Z and entity names a, b, c where $a \lhd person$, $b \lhd room$ and $c \lhd laptop$ we define the following rules:

$$\begin{array}{lcll} a[X] \mid b[Y] & \leftrightarrow & b[a[X] \mid Y] & \text{(walk in/out)} \\ a[X] \mid c[Y] & \leftrightarrow & a[c[Y] \mid X] & \text{(pick up/put down)} \end{array}$$

In plain terms these rules describe how a person may freely walk into and out of rooms and pick up or drop any portable physical objects (represented by entities of sort **laptop**).

For simplicity everything that can happen to a mobile agent (i.e. being created, frozen, defrosted, killed or migrated) is considered as a sequence of primitive operations of the following two types: (*i*) leaving a particular context; (*ii*) entering a particular context. For example an agent creation is considered a single event – the new agent entering its initial context. Killing an agent is a single leaving event. An agent migration from a to b is considered a sequence of two events: (*i*) leaving the source context a; and (*ii*) entering the destination context b. Freezing an agent is considered as a migration into a special context called $frozen$ and defrosting is a migration out again.

To represent the installed security policies, we assume a pair of infix predicates, *can_enter* and *can_leave*, defined later in Sect. 2.8, which for a given agent d and context e behave as follows:

d *can_leave* e holds if the policies allow d to leave the context e
d *can_enter* e holds if the policies allow d to enter the context e

For entities $d \lhd agent$, and $e \lhd context$ we write the rule:

$$e[!d \mid X] \quad \rightarrow \quad e[d[\mathbf{0}] \mid !d \mid X] \quad \textit{iff } d \textit{ can\_enter } e \quad \text{(agent created)}$$

This rule asserts that agents may be created in those places containing an appropriate agent factory (represented by $!d$) provided the new agent is allowed to enter the surrounding context. Similarly, agent destruction is only permitted if the agent is allowed to leave the containing context, as described by this rule:

$$e[d \mid X] \quad \rightarrow \quad e[X] \quad \textit{iff } d \textit{ can\_leave } e \quad \text{(agent killed)}$$

Agents are frozen by moving them into specially created $frozen$ contexts, created dynamically. These contexts are associated with no permissions i.e. $privs(frozen)$ is $\{\}$. Consider the example in Sect. 2.3 – if the `music player` agent were frozen then it would lose the ability to play sound. The acts of freezing and defrosting are described by the following rules:

$$\begin{array}{lcll} e[d \mid X] & \rightarrow & e[frozen[d] \mid X] & \textit{iff } d \textit{ can\_leave } e \quad \text{(agent frozen)} \\ e[frozen[d] \mid X] & \rightarrow & e[d \mid X] & \textit{iff } d \textit{ can\_enter } e \quad \text{(agent defrosted)} \end{array}$$

Note that, to be frozen, an agent must be allowed by the security policies to leave its current context. There is no guarantee the agent will ever be unfrozen again; unfreezing may only occur if the agent has permission to reenter the original context.

Agent migration between contexts is handled by the following rules:

$$\begin{array}{lcll} e[d \mid X] & \overset{emit(d)}{\longrightarrow} & e[X] & \textit{iff } d \textit{ can\_leave } e \quad \text{(agent leaves)} \\ e[X] & \overset{receive(d)}{\longrightarrow} & e[d \mid X] & \textit{iff } d \textit{ can\_enter } e \quad \text{(agent enters)} \end{array}$$

$$\frac{X \overset{emit(\square)}{\longrightarrow} Y \quad Y \overset{receive(\square)}{\longrightarrow} Z}{X \rightarrow Z} \quad \text{(migrate)}$$

Note that the act of migration is a compound operation where the side-effect $emit(\square)$ must be matched by a corresponding side-effect $receive(\square)$. Therefore migration may only happen if the policies allow both the leaving step and the arriving step; it is impossible for the agent to get stuck somewhere in between. It is important to emphasise that only the results of toplevel transitions are visible to applications – applications cannot see any intermediate states of the model. We get away with this because our work so far has focused on a trusted "intranet"-style environment where complications due to unreliable network communication and partial failure are minimised.

Agent migration could be represented differently if we allowed agents to simply climb the entity hierarchy and then walk down again – the approach taken in the Ambient Calculus. This would allow us to simplify our rules by removing the labels on our transition relation. However, allowing an agent to move anywhere in the hierarchy could lead to violations of the sorting rules (described in Sect. 2.2). Additionally there is a subtle semantic difference with respect to the security policies: by using the "teleporting" approach described here, only the configurations at the start (the leaving step) and at the end (the arriving step) are relevant. If the agent were to have to walk from one place to another then the migration could potentially be blocked by a policy attached to an entity somewhere in the middle.

To complete our description of how the model can be updated we have the following rules where X' and Y' are entities:

$$!\square \quad \rightarrow \quad !\square|\square[\mathbf{0}] \quad \textit{iff } \square[\mathbf{0}] \not\lhd agent \quad \text{(non-agent entity created)}$$

$$\frac{X \xrightarrow{\square} Y}{\square[X] \xrightarrow{\square} \square[Y]} \text{ (nested update)} \qquad \frac{X \xrightarrow{\square} Y}{X \mid Z \xrightarrow{\square} Y \mid Z} \text{ (parallel update)}$$

$$\frac{X' \equiv X \lhd X \xrightarrow{\square} Y \lhd Y \equiv Y'}{X' \xrightarrow{\square} Y'} \text{ (update } \equiv)$$

Informally the first rule states that non-agent entities may be created in entity factories (note that agent entities may only be created if allowed by the security policies, using the rule (agent created) described earlier). The other three rules state that transitions may occur anywhere in the entity nesting hierarchy, in parallel with arbitrary other entities up to structural congruence. Note that the labels on the transitions are preserved but must eventually be cancelled further up the tree (by the rule (migrate)).

2.6 Expressing Policies

A security policy is defined as a 4-tuple $\langle location \lhd formula \lhd times \lhd onfail \rangle$ where $location$ is a path expression (see Sect. 2.4) designating a set of specific entities where the assertion given by $formula$ should hold. If, with respect to the time period described by $times$, the assertion becomes violated (e.g. by the physical movement of an object) then the system will attempt to execute the command described in the field $onfail$.

The policy field $times$ can contain one of two possible types of values: $Always(t)$ and $Sometime(from \lhd to \lhd t)$. In both cases the parameter t specifies how much "reaction" time the system has before the policy $onfail$ action is executed. The value $Always(t)$ indicates that the assertion $formula$ should hold for all time during which the system is running. The value $Sometime(from \lhd to \lhd t)$ states that $formula$ should hold[1] at some point in the time interval between the times $from$ and to.

The policy field $onfail$ specifies an action to take should the policy be violated. The action can be of the following types:

[1] This is similar to the concept of *obligation* in traditional Role-Based Access Control (RBAC) systems i.e. it states that someone *should* perform some action during some time interval.

- $Log(message)$ causes a message to be written to a log;
- $Kill(pathexpr)$ asks the system to terminate agents identified by the path expression $pathexpr$;
- $Freeze(pathexpr)$ requests agents named by $pathexpr$ be frozen; and
- $Create(path)$ requests the agent factory named by $path$ create an agent.

For both the $Kill$ and $Freeze$ values we adopt the convention that if the path expression has a missing initial element (i.e. it starts with `/` or `/../`) we automatically prepend the full path to the specific entity the formula is currently being applied to. For example if the policy $location$ field is $a/*$ and the policy is violated at $a \downarrow b$ then the $onfail$ expression $Kill\ /c$ is expanded to $Kill\ \ a/b/c$ i.e. a request to terminate *only* the entity named by $a \downarrow b \downarrow c$ and not any other element (e.g. $a \downarrow d \downarrow c$). This ability to refer to previously matched data in a pattern is also found in other systems using regular expressions, e.g. perl [17].

The policy field $formula$ contains an expression written in a simple spatial modal logic similar to the Ambient Logic [3]. The core syntax is as follows, where $\square$ ranges over entity names:

$formula$	$\leftarrow$	$\mathbf{T}$	(true)
	$\mid$	$\neg formula$	(negation)
	$\mid$	$formula \vee formula$	(disjunction)
	$\mid$	$\mathbf{0}$	(void)
	$\mid$	$\square[formula]$	(named entity)
	$\mid$	$!\square$	(named agent factory)
	$\mid$	$formula \mid formula$	(composition)
	$\mid$	$\Diamond e$	(somewhere modality)

F (false), $a \wedge b$ and $\Box a$ (everywhere modality) may be written using the core syntax as $\neg\mathbf{T}$, $\neg(\neg a \vee \neg v)$ and $(\neg\Diamond\neg a)$ respectively. These constructs may be familiar to those versed in modal logics, but we summarise their meaning in the following section.

2.7 Satisfaction

We say that an entity e satisfies the logical formula f (i.e. the formula f holds at e) by writing $e \models f$. Intuitively, we may think of a formula f as *matching* an entity e if $e \models f$. The relation, $\models$ is defined informally as follows:

- $e \models \mathbf{T}$ for any entity e
- $e \models \neg f$ if $e \models f$ does not hold
- $e \models f \vee g$ if either $e \models f$ or $e \models g$
- $e \models \mathbf{0}$ if e is "nothing"
- $e \models !\square$ if $e \equiv !\square$
- $e \models \square[f]$ if $e \equiv n[M]$ and $\square = n$ and $M \models f$
- $e \models f \mid g$ if $e \equiv N \mid M$, $f \models N$ and $g \models M$
- $e \models \Diamond f$ if $\exists e' \triangleright e \downarrow^* e'$ and $e' \models f$

For example, the formula $\mathbf{0}$ only matches "nothing" (or "void") i.e. the absence of anything. The formula $f \mid g$ matches e if e can be written as the composition of two expressions N and M (remember the equivalence relation $\equiv$) such that f matches N and g matches M. The formula $\Diamond f$ matches e if there is an entity e' somewhere in the tree rooted at e where e' matches f.

2.8 Reasoning about Policies

If we allow individual users to write their own security policies then we must also provide a mechanism to resolve policy conflicts when they arise. Conflicts between rules in our system are similar to those found in Active Databases [5]. Many mechanisms have been proposed, ranging from a simple numeric priority schemes to more complex algorithms comparing rules based on their generality [8] (e.g. the more general rule holds except when the less general does not or v.v.). There is no single best strategy that works perfectly in all circumstances. Our main goal is to make the system be intuitive enough for ordinary users to understand. Security policies in our system are based on a spatial modal logic therefore we also use a spatial mechanism for arbitrating between conflicting policies.

Recall that we model the state of the world as a nested tree of entities (see Sect. 2.2). We observe that within a real life enterprise people too are often arranged into a hierarchy, with the boss at the top, managers in the middle and normal employees at the leaf nodes. In such an organisation, a manager would be able to set a policy which would override those of subordinates but which could itself be overridden by the boss. These two hierarchies, one describing the world and one describing the people, can be linked together by associating entities with a set of people ("owners" or "administrators") via a function

$$owners : entity \rightarrow person\ set$$

such that for an entity e we have $owners(e) = \{person_1 \text{`} \rhd\!\rhd\!\rhd \text{`} person_k\}$ where $person_1 \text{`} \rhd\!\rhd\!\rhd \text{`} person_k$ are the direct "owners" of e. In a typical configuration, the boss would "own" the root entity while normal employees would "own" their individual offices. Our scheme for arbitrating between conflicting policies may be informally described as:

> For a proposed change to entity e, policies instituted by a user $u'' \in owners(e'')$ override those policies instituted by a user $u' \in owners(e')$ where $e'' \downarrow^* e'$ and $e' \downarrow^* e$ as long as $e'' \neq e'$ and $u'' \neq u'$.

Recall from Sect. 2.5 that the installed security policies may be represented by a pair of predicates, *can_leave* and *can_enter* which, given an agent and a context hold precisely when an agent is allowed to leave or enter the context respectively. Both of these predicates are computed in the following way: For a proposed change in the configuration at context c (e.g. an entity wishes to leave c) we first compute the set of users who "own" any of the entities on the path $p_1 \downarrow \rhd\!\rhd\!\rhd \downarrow p_n$ from the "root" entity p_1 which designates c

$$users = \bigcup_{k=1}^{n} owners(p_1 \downarrow \rhd\!\rhd\!\rhd \downarrow p_k)$$

Each user $u \in users$ is allocated a single vote on the proposed change. Note that this effectively means that although users may write policies about entities they do not "own" these policies will be easily overridden by other users who *do* "own" these entities. A

user u votes for the proposed change if the number of their policies which are in violation decreases, votes against if the number in violation increases and abstains otherwise. We define a function $vote(user)$ as follows:

$$vote(user) = \begin{cases} -1 \text{ if } user \text{ votes against the proposal} \\ 0 \quad \text{if } user \text{ abstains} \\ +1 \text{ if } user \text{ votes for the proposal} \end{cases}$$

We then compute the value of

$$overall\ vote = \sum_{i=1}^{n} \sum_{o \in owners(p_1 \downarrow \bowtie\bowtie \downarrow p_i)} prio(i) vote(o)$$

where $p_1 \downarrow \bowtie\bowtie \downarrow p_i$ refers to the ith entity on the path $p = p_1 \downarrow \bowtie\bowtie \downarrow p_n$ and the function $prio(i)$ gives the priority of owners of this entity. One possible priority function is given by $prio(i) = x^{-i}$ where x is a tunable vote weighting factor. The parameter x determines how many people who "own" an entity p_n are needed in order to equal the vote of a single person who "owns" a "more important" entity p_{n-1}. If $x > max_i\ (|owners(p_i)|)$ then it is impossible for the owner of a more important entity to be overridden by a group of people who own a less important entity. The system will allow the proposed change if $overall\ vote \geq 0$ and veto it otherwise.

3 Policy Examples

In this section we demonstrate the kinds of policies which are expressible in our system by means of a series of examples set in a typical shared workplace environment. A snapshot of the world configuration is presented in Sect. 2.3. The top-level entity is named `World` and contains child entities `Bob's office` and `Charlie's office` representing the offices of users named Bob and Charlie respectively. We assume that ordinary employees by default "own" the entities corresponding to their offices and for the sake of an interesting example we further assume that Bob is the boss and also "owns" the top-level entity, `World`.

A user, called Alice, writes and deploys a "follow-me" music playing mobile agent which follows her around, playing music where she goes. She is worried about the agent running amok and so writes the following policy to enable the system to monitor the agent:

$$\begin{array}{lcll} \langle & location & = & \texttt{World}^{\mathsf{c}} \\ & formula & = & \Diamond(\text{Alice}[\mathbf{T}] \mid \Diamond\texttt{music player}[\mathbf{T}] \mid \mathbf{T})^{\mathsf{c}} \\ & times & = & Always(10\ seconds)^{\mathsf{c}} \\ & onfail & = & Log \quad \rangle \end{array} \tag{1}$$

"for all time, wherever in the `World` I am, an agent called `music player` should be in the same space as me. If this is not true for more than 10 seconds, log the error"

Remember that $e \models f \mid g$ holds whenever f and g are children of e and that $\mathbf{T}$ matches anything, including $\mathbf{0}$, the absence of anything. In the formula above the third $\mathbf{T}$ means that the formula will hold irrespective of whatever else is in the same space as Alice.

The consequences of this policy are summarised as follows:

1. When the `music player` attempts to migrate, the system prevents the agent from *leaving* the same room as the user. Note it does not directly force the agent to move properly, it just stops it from moving inappropriately.
2. Upon observing Alice move to a new room the system assumes the agent is broken if it has not followed her within 10 seconds. The system will log the error for Alice to use in debugging her errant agent.
3. If Alice moves to a room which already has a `music player` agent the system will not complain even if Alice's agent fails to follow her.

Consider a second user, Bob, who is Alice and Charlie's boss. Bob prefers peace and quiet where he works. To prevent wandering music playing agents disturbing him he writes a rule:

$$\begin{array}{lcll} \langle & \mathit{location} & = & \texttt{World/*}^{\frown} \\ & \mathit{formula} & = & \Box\neg\texttt{Bob}[\mathbf{T}] \vee (\Diamond\texttt{Bob}[\mathbf{T}] \wedge \Box\neg\texttt{audio}[\neg\mathbf{0}])^{\frown} \\ & \mathit{times} & = & \mathit{Always}(3\ \mathit{seconds})^{\frown} \\ & \mathit{onfail} & = & \mathit{Freeze}\ \texttt{/.../audio/*} \quad \rangle \end{array} \tag{2}$$

"if ever I'm in an office with a music playing agent, freeze the agent if it has not left within 3 seconds"

The policy *location* field `World/*` causes the rule to be applied to all children of the entity named `World`, i.e. in the diagram in Sect. 2.3 this corresponds to all the offices, `World` $\downarrow$ `Bob's office` and `World` $\downarrow$ `Charlie's office`. The same formula is applied individually to each of these entities. The formula $\Box\neg\texttt{Bob}[\mathbf{T}]$ holds if the entity `Bob` is nowhere inside the office; the formula $\Diamond\texttt{Bob}[\mathbf{T}]$ holds if the entity `Bob` is *somewhere* inside the office and the formula $\Box\neg\texttt{audio}[\neg\mathbf{0}]$ holds if there is not a non-empty `audio` context anywhere within the office. Taken together, the whole *formula* may be read as

Either Bob is not inside the office concerned (in which case there is no violation) or he is inside the office but there is no sound playing.

If the policy is violated in the office named x then the onfail action is expanded to *Freeze* `World/`x`/.../audio/*` causing audio playing agents inside office x to be frozen.

The consequences of this policy are summarised as follows:

1. If a `music player` agent attempts to migrate inside the same office as Bob the request will be denied, assuming that his policy is not overridden by anyone more senior in the company.
2. If a `music player` agent running on a laptop or PDA is physically moved inside his office by someone else, that agent will be frozen.

Now consider what will happen when Alice enters Bob's office. Clearly the two policies 1 and 2 now conflict. Alice's mobile agent will attempt to migrate inside Bob's office so the system will apply the conflict resolution rules described in Sect. 2.8. Assuming the system knows that Bob "owns" the entity named `World` (since he is the boss) his policy will override those belonging to Alice and the migration request will be blocked.

Imagine a third user, Charlie, with more malicious intent. This user attempts to lure hapless agents into his domain and then trap them there forever. He decides to go after Alice's music playing agent and writes the following:

$$\begin{array}{lcll} \langle & \mathit{location} & = & \texttt{World/Charlie's office}\text{‹} \\ & \mathit{formula} & = & \Diamond(\texttt{music player}[\mathbf{T}])\text{‹} \\ & \mathit{times} & = & \mathit{Always}(0\ \mathit{seconds})\text{‹} \\ & \mathit{onfail} & = & \mathit{Log} \qquad\qquad \rangle \end{array} \tag{3}$$

"for all time the `music player` agent should remain inside my office."

Consider what happens when Alice is enticed into Charlie's office for a coffee and biscuit. Initially Alice's `music player`'s request to migrate into Charlie's office is accepted since it does not violate any policy (in fact it causes rule 3 to no longer be in violation – an improvement!) When Alice leaves the office the music player attempts to follow her. Charlie's and Alice's rules are now in direct conflict. Unfortunately for Alice since Charlie "owns" his office his policies take priority and therefore the agent's request to leave is denied. What can Alice do? The only solution for Alice in this situation is to appeal to a higher authority – in this case Bob – someone whose policies are ranked higher than Charlie's. Bob may write a policy to evice Alice's agent, overriding Charlie's wishes.

4 Related Work

There have been many proposed mobile agent systems, e.g. TACOMA [12] (Tromsø And COrnell Moving Agents), Agent-TCL [6] and Telescript [16]. Similarities with our work include: mobile agents on mobile devices (PDAs [9] and mobile phones [10]) in TACOMA and the concept of *regions* (similar to our *entities*) in Telescript. Unlike our work, none of these previous systems attempted to exploit spatial modal logic to bridge the gulf between the physical world of people and the virtual worlds of mobile agents.

Jiang and Landay [11] consider risks to privacy in context-aware systems. They base their work on the abstraction of *information spaces*, similar to our *entities*. They envisage a system where documents have associated *privacy tags* which are used to prevent the unwanted leakage of data. IBM Aglets [13] provide a Java-based API for building mobile agents. Their security mechanism is based on the Java-2 security model: code is selectively trusted or not depending on its origin and/or the presence or absence of signatures. The LocALE (Location-Aware Lifecycle Environment) framework provides a CORBA-based mechanism to control the life-cycle (i.e. creation and destruction) and location of software objects residing on a network. LocALE defines the notion of a *Location Domain* – a group of machines physically located in the same place. The

difference between all three of these systems and our work is that none of them allow the specification of spatial mobility security policies.

Our work is inspired by the theoretical work on the Ambient Calculus [2] and the Ambient Logic [3]. Although our model is a great deal simpler than that proposed in the Ambient Calculus, it still allows us to combine together the physical world of people and the virtual world of mobile agents into a single, unified representation. The subset of the Ambient Logic used in our policy definitions remains computationally decidable and simple for humans to understand while still allowing the a great deal of flexibility and expressiveness.

5 Conclusion and Future Work

We have presented a technique for expressing spatial (i.e. location-based) security policies for mobile agents. These policies can be used to make both positive and negative assertions about the dynamic location of agents. Assertions may refer to the location of both physical and virtual objects in the world, a feature useful for location-aware Sentient applications. This technique provides a useful way to constrain the mobility of agents, to use mobile agent technology safely and to simplify the development process of future Sentient applications.

Our work was inspired by recent theoretical work on mobile computation, specifically the Ambient Calculus [2] and the Ambient Logic [3]. In future we would like to investigate how we could enhance our model of the world by adding in Ambient Calculus-style process expressions representing agents. Entities like people could be represented by mobile agents which have the capability to move anywhere at any time. The process expression associated with a mobile agent could be considered a characterisation of its behaviour – a contract with the system – which could be checked for consistency with global policy before the agent is allowed to run.

It is currently possible to do a small amount of up-front static checking of security policies: policies about locations which are known to be fixed can be checked at policy-install time. However, since we only have limited control over the physical environment we cannot do much about a policy which says, "the company laptop never leaves the building". Clearly this policy could be violated by an individual picking up the laptop and walking home with it.[2] Additionally, some agents may wish to perform limited checking of policies at runtime. For example Alice's `music player` agent from Sect. 3 may pose the question "if I enter Charlie's office, will I definitely be allowed to leave?" in an attempt to avoid being trapped.

Perhaps the most interesting avenue for future work is to investigate how to scale the system up beyond one organisation. Our model of the world assumes that everything can be arranged in a single hierarchy, with a world controller in absolute control of everything. This approach might work adequately for a small organisation but to scale any further we need to cope with a multitude of problems: unreliable wide-area network communication, mutual distrust between organisations etc. One possibility is to employ

[2] We could envisage a system in which the doors are under our control and can be locked to prevent the laptop leaving. However consider that Health and Safety legislation would require the doors to automatically unlock if there was a fire!

a two-level approach where within an "intranet" agents are managed using this system while the "internet" case is handled differently.

A further enhancement to this work is to provide support for multiple simultaneous parallel hierarchies, allowing the same object to exist in several places at once. This facility could be used to represent different "views" of the same environment (for an analogy consider that a user may be present and active in more than one Internet "chat-room" simultaneously). When an object moves in one world they may also have to move in another. Reconciling these parallel views is an interesting topic of future work.

In summary, based on the research described in this paper we claim that our work provides a strong foundation for the building of Sentient, location-aware applications with a basis in theory. We believe that Sentient environments are an interesting niche for applications developed using mobile agent technology and our models help design these applications in a less ad-hoc manner.

Acknowledgements. This work was supported by the Schiff Foundation and the Engineering and Physical Sciences Research Council (EPSRC) and sponsored by AT&T Laboratories Cambridge Ltd. The authors would like to thank Richard Sharp for valuable comments and Andy Hopper for all his advice and support.

References

1. Luca Cardelli. Semistructured Computation. In *Proceedings of 7th International Workshop on Database Programming Languages, DBPL'99*, Kinloch Rannoch, Scotland, UK, September 1999.
2. Luca Cardelli and Andrew D. Gordon. Mobile Ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 1378, pages 140–155. Springer-Verlag, 1998.
3. Luca Cardelli and Andrew D Gordon. Anytime, Anywhere Modal Logics for Mobile Ambients. In *Principles of Programming Languages (POPL)*, 2000.
4. World-Wide Web Consortium. XML Path Language (XPath) Specification, November 1999. http://www.w3.org/TR/xpath/.
5. Umeshwar Dayal, Eric N. Hanson, and Jennifer Widom. Active database systems. In *Modern Database Systems*, pages 434–456. 1995.
6. R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In M. Diekhans and M. Roseman, editors, *Fourth Annual Tcl/Tk Workshop (TCL 96)*, pages 9–23, Monterey, CA, 1996.
7. Andy Hopper. 1999 Sentient Computing. *Phil. Trans. R. Soc. Lond.*, 358(1):2349–2358, 2000.
8. Yannis E. Ioannidis and Timos K. Sellis. Conflict resolution of rules assigning values to virtual attributes. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, 1989.
9. Kjetil Jacobsen and Dag Johansen. Mobile Software on Mobile Hardware – Experiences with TACOMA on PDAs. Technical report, Department of Computer Science, University of Tromsø, Norway, 12 1997.
10. Kjetil Jacobsen and Dag Johansen. Ubiquitous Devices United: Enabling Distributed Computing Through Mobile Code. In *Proceedings of the Symposium on Applied Computing (ACM SAC'99)*, February 1999.

11. Xiaodong Jiang and James A. Landay. Modeling Privacy Control in Context-Aware Systems. *IEEE Pervasive Computing magazine*, 2002.
12. Dag Johansen, Robbert van Renesse, and Fred B Schneider. An Introduction to the TACOMA Distributed System Version 1.0. Technical report, Department of Computer Science, University of Tromsø, Norway, 6 1995.
13. Danny B. Lange and Mitsuru Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison Wesley, 1998.
14. K. Schelderup and J. Olnes. Mobile Agent Security — Issues and Directions. *Lecture Notes in Computer Science*, 1597:155–167, 1999.
15. David Scott, Alastair Beresford, and Alan Mycroft. Spatial policies for sentient mobile applications. Draft Manuscript. Available on web page http://www.recoil.org/ djs/papers/spatial02.html, December 2002.
16. J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon '96*, pages 58–63, 1996.
17. Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, 1992.
18. Mark Weiser. The Computer for the 21st Century. *Scientific American*, 9 1991.

Towards UML-Based Formal Specifications of Component-Based Real-Time Software

Vieri Del Bianco[1,2], Luigi Lavazza[1,2], Marco Mauri[2], and Giuseppe Occorso[3]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione
P.zza Leonardo Da Vinci, 32, 20133 Milano, Italy
{delbianc,lavazza}@elet.polimi.it
[2] CEFRIEL, Via Fucini, 2, 20133 Milano, Italy
mmauri@cefriel.it
http://www.cefriel.it
[3] Technology REPLY, Via Ripamonti, 89, 20139 - Milano, Italy
g.occorso@reply.it

Abstract. UML-RT is achieving increasing popularity as a modeling language for real-time applications. Unfortunately UML-RT is not formally well defined and it is not well suited for supporting the specification stage: e.g., it does not provide native constructs to represent time and non-determinism. UML+ is an extension of UML that is formally well defined and suitable for expressing the specifications of real-time systems (e.g., the properties of a UML+ model can be formally verified). However, UML+ does not support design and development. This article addresses the translation of UML+ into UML-RT, thus posing the basis for a development framework where UML+ and UML-RT are used together, in order to remove each other's limitations. Specifications are written using UML+, they are verified by means of formal methods, and are then converted in an equivalent UML-RT model that becomes the starting point for the implementation.

1 Introduction

Formal methods have demonstrated to be effectively applicable in the industrial development of real-time safety critical systems. Nevertheless, formal methods are not widely used in industry. The problem is that while the demand for real-time software increases very fast, the availability of developers who can master formal methods remains little. The consequence is that formal methods are generally considered too difficult or too expensive to be used in "ordinary" real-time software development. On the contrary, UML [14] has achieved a great popularity, essentially because it is a semi-formal notation relatively easy to use and well supported by tools.

Interestingly, UML is gaining popularity also for real-time developments. In fact, UML for Real-Time (alias UML-RT) has been defined on the basis of ROOM [18] and has been rapidly adopted by many developers: it is likely that OMG will include

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 118-134, 2003.

UML-RT features in the definition of UML 2.0. However, the application of UML-RT to the real-time domain is still suffering from several problems:

- UML-RT is not formally well defined. This is a relevant limitation of UML-RT, since very often real-time applications are also safety-critical, and thus call for activities like the verification of properties (such as safety, utility, liveness, …), the simulation of the system, the generation of test cases, etc. It is very hard – if at all possible – to carry out such activities when the specifications are written in semi-formal notations like UML or UML-RT.
- UML-RT is an effective notation for the design and implementation of systems, but not very well suited for representing requirements or specifications. For instance, when modeling the environment in which a real-time system has to work it is often necessary to represent non-deterministic behavior or simultaneous events. These phenomena are not supported by UML-RT.
- Finally, time issues (i.e., the representation of time and time constraints) are not treated at a native level: ad-hoc components (like timers) have to provide time-related information to the system. This is not generally perceived as a big problem at the design level, as designers consider quite natural to model the existence of timers and similar objects.

In previous work we addressed some of the above problems. In fact we adopted a dual language approach to real-time software development: in a first phase models are written in UML according to the usual modeling practices; in a second phase UML models are automatically translated into one or more formal notations, which provide support to activities such as the simulation, the verification of properties, the generation of test cases, etc. In this way, developers exploit the advantages of formal notations while skipping the complex and expensive formal modeling phase, since they can use the notation they are familiar with.

Actually we had to extend UML in order to let it satisfactorily specify real-time systems and to provide it with formal semantics [11,5,7,6]. The models written in the resulting language (called UML+ throughout this paper) can be automatically translated into equivalent TRIO temporal logic formulas [9] or into timed automata [1]. In this way existing formal methods can be applied. For instance the properties of the model can be verified by means of the Kronos model checker [20].

Although such work demonstrated the viability of the approach, our approach did not support the design and implementation activities. Consistently with our goal of defining a development method that can be effectively applied in industrial settings, we chose UML-RT as a target language. The reasons for this choice are that UML-RT is a very good notation for component-based developments, it is going to become a standard (it is very likely that it will be included in UML 2.0) and it is already very popular in industry. The aim of the work reported here is thus to build a bridge between our specification-oriented UML+ and the design-oriented UML-RT.

The paper is structured as follows. Section 2 provides a brief introduction to UML+. Section 3 briefly recalls the main characteristics of UML-RT. Section 4 illustrates the problems for translating UML+ models into UML-RT and describes the proposed solutions. Section 5 describes the development environment that implements

the proposed approach. Section 6 presents a simple case study as a proof of concepts. Finally, Sect. 7 draws some conclusions and briefly accounts for related work.

2 A Brief Introduction to UML+

The semantics associated with UML+ is directly inspired by the Timed Statecharts defined by Kesten and Pnueli [11]. This formalism extends the traditional statecharts by specifying time limits for the execution of transitions. The semantics is defined with reference to a dense time domain. This implies that the system may deal with events that are arbitrarily close in time to each other. Transitions are classified in two types: immediate ones and timed, or waiting, transitions. Immediate transitions do not depend on time: they are executed when a triggering event occurs. When no immediate transition is enabled, the time can flow with the state of the system remaining unchanged. On the contrary, timed transitions are independent from events. They are associated with a time interval that specifies a minimum and a maximum waiting time: the transition cannot be executed before the minimum or after the maximum waiting time. If no event changes the current state, the timed transition must be executed before the maximum waiting time. In Timed Statecharts negated events can appear in the conditions which guard the execution of transitions. The concept of "step" is associated with the execution of an immediate transition; a reaction to an event may occur several steps after its generation, but still in the same timestamp. This kind of semantics is based on the fact that every generated event "persists" until the time does not flow. The time may flow only if all the transitions which were enabled by that event have been executed. In this way, several transitions triggered by the same event *e* are executed before the time becomes grater than the time of the occurrence of *e*.

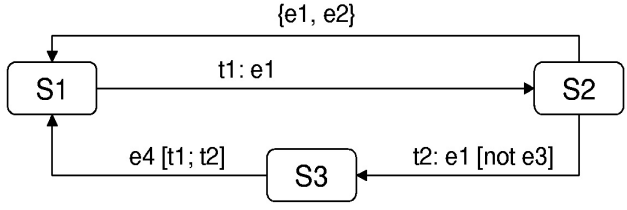


Fig. 1. A UML+ statechart

Timed Statecharts have been further extended in UML+ to accommodate most of the syntax of UML State Diagrams that is not present in Timed Statecharts; for instance, inter-level transitions and fork transitions are allowed in UML+. In UML+ it also possible to associate transitions with both events and time intervals (see for instance the transition from S3 to S1 in Fig. 1).

UML+ allows the modeler to associate a set of events to a transition, indicating that the transition is triggered by the concurrent occurrence of the set of events (see the transition from S2 to S1 in Fig. 1).

In UML+ guards can make reference to events. It is possible, for instance, to specify that a given transition is executed if event *e1* occurs while event *e3* is not occurring (see the transition from S2 to S3 in Fig. 1).

Besides the modification of statecharts, UML+ does not exhibit differences with respect to standard UML. However, our approach to verification of models based on UML+ only takes into account class diagrams, state diagrams and object diagrams. Other diagrams are ignored if present. The elements of the class diagrams are employed with their usual role and meaning. However, it must be noted that – being our approach devoted to the verification of real-time properties only – the attributes and the methods that do not affect real-time behavior of the models are ignored.

A detailed definition of UML+ can be found in [12,5,7].

3 A Brief Introduction to UML-RT

UML-RT is an extension of UML that addresses real-time issues. It provides a formalism to handle active objects. An active object is called a Capsule in UML-RT and it communicates with other capsules through asynchronous messages, which are sent and received through Ports. A Port is defined by a Protocol that defines which messages can be sent through a port (*Out messages*) and which messages a port accepts (*In messages*). Given a Protocol, its conjugate is always defined by simply inverting the In and Out messages.

The State Diagrams associated to each capsule have the usual syntax and semantic as in plain UML, including the "run to completion" behavior [14]. The only additional constraint is that any message (except internal messages, which remain in the boundaries of the State Diagram) is always associated with a Port.

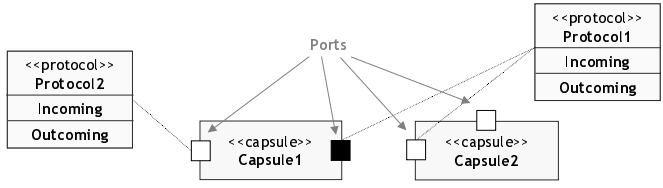


Fig. 2. Capsules and ports in UML-RT

Capsules are connected through Connectors. A Connector binds two different Ports with compatible Protocols. A protocol is always compatible with its conjugate. Moreover, a protocol $\mathcal{P}1$ is compatible with another protocol $\mathcal{P}2$ if $\mathcal{P}1$ accept as In messages a superset of the $\mathcal{P}2$'s Out messages and the set of $\mathcal{P}1$'s Out messages is a subset of the messages accepted by $\mathcal{P}2$.

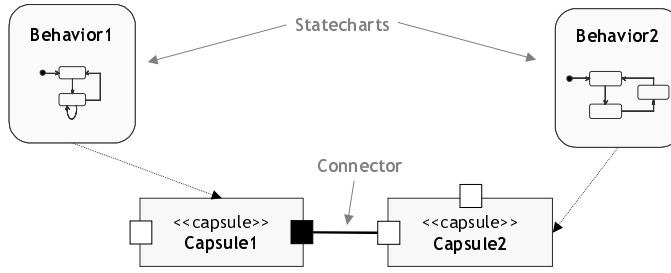


Fig. 3. Connectors and state diagrams in UML-RT

UML-RT mainly focuses on the concept of active component (the Capsule) and does not directly address real-time constraints. The concept of time can be found in standard UML-RT libraries – mainly thanks to the Timer class stereotype– but not directly in the modeling language.

UML-RT is implementation-oriented: it is conceived to be used with a complete library in a language of choice, usually abstracting form the underneath platform. By embedding code fragments in transitions and states (as defined in plain UML) a UML-RT model can be directly translated into code (Rational Rose RealTime being the reference tool for generating working embedded distributed systems from UML-RT models).

4 From UML+ to UML-RT

UML+ is a formal notation to express the desired behavior and the constraints of real-time systems; i.e., it can be used to formalize real-time requirements. UML+ models are written in a visual language very close to standard UML 1.4, and can be translated in a formal language and checked to verify that the systems behave as required and that the constraints are satisfied.

4.1 A Case Study

In the rest of the paper an example is used to illustrate the proposed approach. The system to be modeled and developed is the Generalized Railroad Crossing (GRC) [10], one of the best known benchmarks proposed in the literature for evaluating formalisms and tools dealing with real-time software.

The system to be modeled operates a gate at a railroad crossing. The railroad crossing I lies in a region of interest R (see Fig. 4). Trains travel through R on K tracks in one direction (having trains traveling in both directions does not change the complexity and

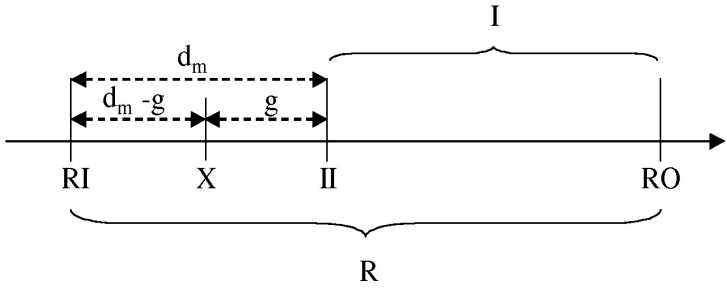


Fig. 4. GRC regions of interest

relevance of the case study). Trains can proceed at different speeds, and can even pass each other. Only one train per track is allowed to be in R at any moment. Sensors indicate when each train enters and exits regions R and I. Point RI and RO indicate the position of the entrance and exit sensors for region R. II indicates the position of the sensor which detects trains entering region I. dm and dM are the minimum and maximum time taken by a train to cross RI-II zone. hm and hM are the minimum and maximum time taken by a train to cross zone I. g is the time taken by the bars of the gate to move from the completely open to completely closed position.

4.2 UML+-RT: Making UML+ Component-Aware

In order to support component-based development we have to make UML+ component-aware. For this purpose we adopt the representation of components proposed by UML-RT, i.e., the capsules. This choice was taken because capsules are a satisfactory formalism, and because in this way it is easier to convert UML+ models into UML-RT models. The result of merging UML+ (featuring the Timed Statecharts [11]) and UML-RT is a new modeling language called UML+-RT.

UML+ specifications of real-time system are essentially composed of class diagrams and object diagrams; the instances of classes are active objects and their behavior is defined by the associated Timed Statecharts. The structure of the system is defined by an Object Diagram that is used to connect the class instances that compose the system. In UML+-RT we retain the same organization. As an example, let us consider the model of the GRC. Figure 5 illustrates the class diagram of the system, where capsules, ports and protocols are explicitly modeled. The object diagram is used to define the first level capsule. This is relatively straightforward, since the Component Diagram of UML-RT is a specialization of the UML Collaboration Diagram [16,18], as are the object diagrams. Figure 6 illustrates the structure of the railroad crossing system in terms of capsules and connections, highlighting the connections to ports.

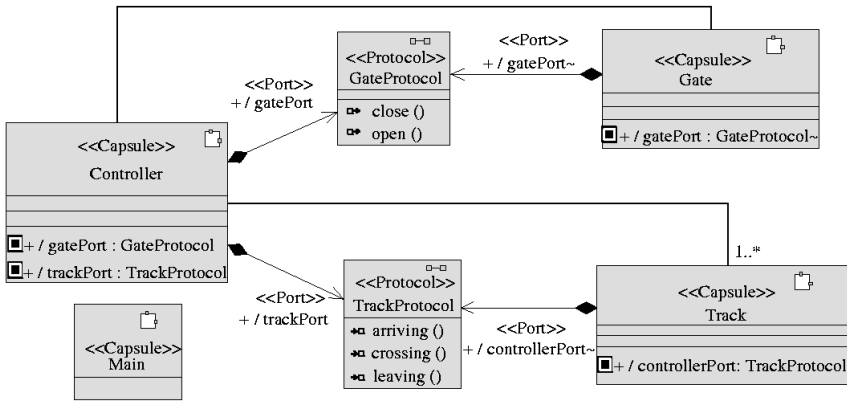


Fig. 5. UML+-RT class diagram

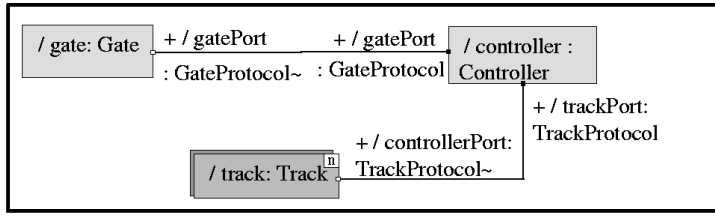


Fig. 6. Main capsule: UML+-RT component diagram

The behavior of a Capsule is defined by the associated Timed Statechart. For instance, Fig. 7 describes the behavior of class Gate: it clearly indicates that the events which cause the state transitions are received through the gatePort.

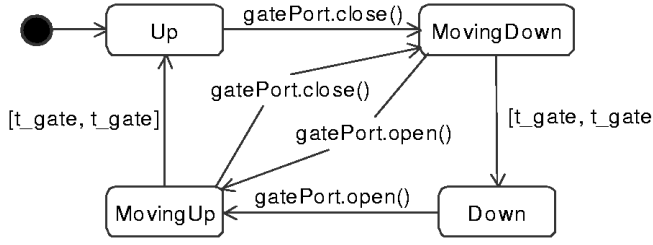


Fig. 7. Timed statechart for the Gate class

4.3 Dealing with Synchronous Semantics

UML+ adopts the synchronous semantics of Timed Statecharts. Synchronous semantics – although useful in the analysis phase and ideal for the verification by model checkers – is unrealistic in the implementation phase. In a real system there are no cheap and easy ways to verify that two events are concurrent, to make two transitions fire simultaneously, and when the system is distributed it is not even easy to process messages in the same order in which they were generated. On the contrary, systems are implemented according to an asynchronous semantics. Most of the modeling tools that generate an implementation are based on asynchronous models. Hence it would be desirable to be able to transform UML+ synchronous specifications into asynchronous models, like UML-RT models. UML-RT was chosen as the target notation because it is probably going to become a standard, it is component-oriented (an important feature to facilitate reuse), and it is equipped with tools (one for all, Rational Rose RealTime) to generate working systems.

The translation from UML+-RT to UML-RT is not simple. The "run to completion" semantics of the UML-RT State Diagrams is quite different from the "synchronous execution of transitions" semantics of Timed Statecharts adopted in UML+ and UML+-RT. This difference can lead to some hard problems, described in Sect. 4.4. The problems that can arise are of two types:

- Some model fragments simply do not make sense in UML-RT. For instance, in UML-RT it is not possible to deal with negated events or to associate a transition with a set of simultaneous events.
- Some models have the same general meaning in UML-RT as in UML+, but their behavior is not actually the same.

In the first case it is difficult to devise which UML-RT model would more closely represent the intended meaning of the given UML+ model. Therefore we decided not to translate models having this kind of characteristics, i.e., it is responsibility of the modeler to correct these situations.

In the second case the model interpreted according to the UML-RT semantics could violate the properties that were proved valid for the *same* model interpreted according to UML+ semantics. In this case the modeler is invited to build a second, more realistic UML+ model that takes into account the execution environment, and is therefore able to predict whether properties will remain valid also in the implementation-oriented UML-RT model.

The next section describes the problems mentioned above (and how to deal with them).

4.4 Issues and Problems with the Translation from UML+-RT to UML-RT

Consider the transition from S2 to S1 and the transition from S2 to S3 in Fig. 1: managing concurrent events and negated events in UML-RT is impossible, therefore in

these cases the translator issues a warning and does not produce any UML-RT model. It is the modeler who has the responsibility to produce a more realistic model having similar characteristics. For instance the transition from S2 to S1 could be modified as shown in Fig. 8.

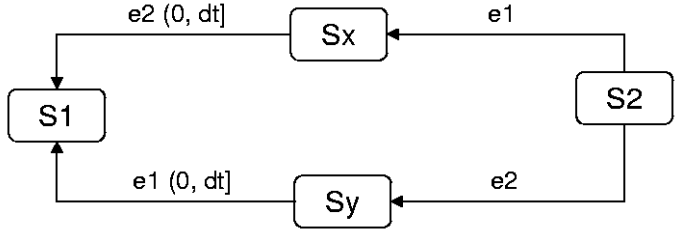


Fig. 8. A statechart handling "almost concurrent" events

The statechart in Fig. 8 prescribes that the transition from S2 to S1 happens when events e1 and e2 occur (in any order) in a [0, dt] interval. If dt is little the transition will happen when e1 and e2 occur in a very short – though finite – interval, while the original condition (given in Fig. 1) required that e1 and e2 occurred exactly at the same time. In practice there is generally no difference between the two specifications. In any case it is always possible to model-check the new version of the model containing the transition from S2 to S1 redefined as shown in Fig. 8, in order to assure that the desired properties still hold.

Transitions involving negated events, timed transitions having $t_{min} = t_{max}$ (i.e., transitions reacting to events that must occur at a precise time) and instantaneous transitions (i.e., transitions that take null time to execute, even though they are associated with some action), can be treated in a similar way. In particular, we can specify that two events do not occur in a short finite interval, or that an event occurs in a short finite interval, or that an action takes a short finite time to complete.

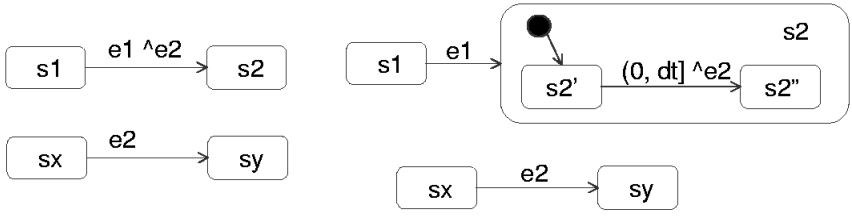


Fig. 9. A model containing simultaneous events (left) and the equivalent model with no simultaneous events

Timed transitions, i.e., transitions bounded with a time interval $[t_{min}, t_{max}]$, are easily represented in UML-RT when $t_{max} > t_{min}$. When a state having one or more of such outgoing timed transitions is entered clocks are set (via the UML-RT libraries) to t_{min}. When the state completes its activities (if any), it waits the expiration of the appropriate clock before evaluating the guards associated with the timed transitions, and finally

fires one of the enabled transitions. Clocks can be set also on the upper bound t_{max}. In this case, if the state completes its activities after the upper bound, an exception is raised, reporting that the state activities have not respected the timing constraints specified in the original UML+-RT model. These exceptions are useful for testing; in any case it is a design decision how to handle them, i.e., whether to execute a transition even if the upper bound time has passed or to take a different action.

Now let us consider the statechart reported in the left part of Fig. 9. In this case the statechart has a well defined meaning in UML-RT, but unfortunately the behavior of the system in UML-RT is not the same as in UML+. In fact in UML+ events e1 and e2 occur at the same time (i.e., in the same timestamp), while in UML-RT e2 will follow e1 with a finite (though probably little) delay, since events are extracted from the event queue one at a time. This means that the properties that hold for the UML+ model could be violated by the *same* model behaving according to UML-RT semantics.

In these cases we have models that are easily translated into UML-RT, but whose behavior will not match the previously model-checked one. We decided to solve this problem by maintaining two models: one for the purpose of translation, and another for the purpose of model checking (see Sect. 5). The latter model must reflect the behavior of the target UML-RT model. This can be achieved in two ways:

1. The modeler explicitly represents that transitions take a non-null time, e.g., by assigning a positive lower bound to the time intervals associated with the transitions. For instance, the UML+ model reported in the right part of Fig. 9 behaves as the UML-RT model reported in the left part of the same figure: event e2 follows event e1 after a finite time (not grater than dt).
2. The modeler explicitly models in UML+ the event queue that is implicitly assumed by UML-RT statecharts. This requires to set the maximum size of the queue.

In both cases the size of the model increases very fast as the number of events that have to be taken into account in the realistic delay period increases.

Figure 10 illustrates a model exploiting a queue. The queue is a FIFO container that is loaded with the arriving events and releases them in a finite time, which represents the time actually needed to perform a transition. In the left part of Fig. 10 when e1 (or e4) occur, instead of issuing the events e2 (or e3) which would imply an instantaneous transition to state sy (or sz), the events $e2_q$ (or $e3_q$) are issued. These events are "captured" by the queue (right part of Fig. 10), which will issue the event e2 (or e3) after a finite delay ≤ dt, thus causing the transition to sy (or sz). Note that here we made two assumptions:

- The maximum time to handle a transition is dt. Therefore the transition is associated with a time interval (0,dt]. If a minimum time dt_min were also defined the time interval would be [dt_min,dt].
- The length of the queue is 1. In general the length of the queue is given by the maximum number of events that can arrive during time interval [0, dt]. In the case depicted in Fig. 10 e2 and e3 are considered mutually exclusive (at least in the considered situation, i.e., immediately after the transition from s1 to s2 or s3).

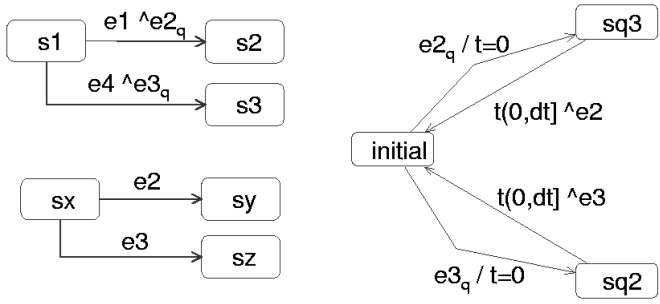


Fig. 10. A model featuring a queue

In Fig. 10 the notation t(0-dt] associated with a transition indicates that the transition has to occur in the interval (t,t+dt]. The notation t=0 indicates that clock t is reset. In this simple case this notation is actually not necessary (since t is always reset when exiting from state initial, t(0-dt] is equivalent to (0-dt] in the transitions from sq2 and sq3 towards initial). However, in longer queues it is necessary to use this notation to express properly the time constraints.

5 The Development Environment

Thanks to the adoption of Timed Statecharts, UML+-RT has well-defined semantics. This allowed us to write a program that translates UML+-RT models into timed automata, so that the model checking tool Kronos [20] can be used to verify properties [3]. This is sufficient to support the specification phase, but does not help much in the design and implementation phase. For this purpose we want to generate a UML-RT model that preserves the properties already formally verified. However, the original model has to be modified, since it features perfect concurrency or synchronicity, which cannot be generally implemented in a model featuring asynchronous semantics. The problem is to preserve the properties of the original model, but also to check that these properties do not rely on characteristics of the model that cannot be implemented in the real world. Our approach is to make incremental refinements of a UML+-RT model until we achieve a model that can be safely translated into UML-RT. At first sight, this approach could seem to be a not very good trade-off, as long as most of the difficulties of the conversion are left with the modeler. However we believe that in this way some relevant advantages are achieved:

- the modeler is guaranteed that the properties of the original model are maintained;
- the modeler is guided in the modification of the model by the diagnostic messages provided by the translator;
- the modeler achieves a deeper understanding of the model, as new details (like the realistic duration of a transition) are added;
- the final model actually reflects the ideas of the modeler.

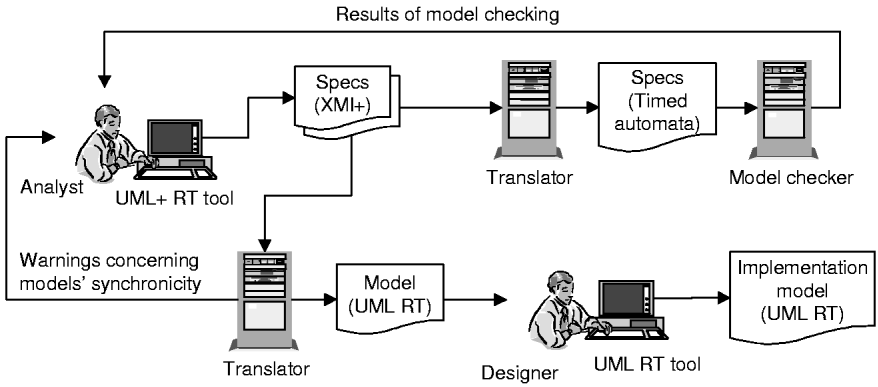


Fig. 11. The envisaged programming environment

Figure 11 illustrates the programming environment that implements the proposed approach. This environment supports a development process organized as follows.

Initially the modeler creates a UML+-RT model without worrying about the implementation of the model. The product of this phase is an "ideal" model, whose properties are verified by means of the model checker.

The ideal model is then processed by the translator, which tries the conversion into UML-RT. In general the translator will find problems with the synchronicity of the model (see Sect.4.4) that are notified to the modeler. The latter modifies the model in order to remove the most obvious problems (simultaneous events, negated events, etc.). The new model is again verified by means of the model checker.

The resulting "more realistic" model M is finally translated into a UML-RT model M_{RT}. This step is mainly devoted to replacing timed transitions with transitions triggered by events generated by timers; the setting, resetting, etc. of timers is determined by the translator on the basis of the timed transitions. However, M_{RT} will not behave exactly like M, e.g., because in M transitions are instantaneous, while in M_{RT} they take a finite time. In order to verify the properties of M_{RT} the modeler builds a second UML+-RT model M', which modifies M in order to represent its behavior according to the rules of the UML-RT environment (e.g., by introducing suitable queues of events). In practice in this phase the modeler maintains two models: M' for the purpose of model checking, and M for the purpose of translation into M_{RT}. The behavior of M_{RT} in the UML-RT environment is equivalent to the behavior of M' in the synchronous model-checking environment. This equivalence descends from the fact that the modeler built M' as a version of M where the "non-ideality" of the execution environment is *adequately* considered. We considered the automatic construction of M' too complex and/or too impractical to be attempted. The result is that the responsibility of maintaining M and M' equivalent remains with the modeler. Generally, he/she is in a very good position to judge which features of the model M are too idealized, and, in these cases what are the sufficiently realistic corresponding models. For instance, the modeler can evaluate if a delay in handling a signal is long enough to alter the behavior of the system, and – if so – to explicitly model that delay (note: a prudent approach is to model the delay, and then remove it if the behavior is not affected).

At the end of this process, M_{RT} is the starting point for the coding and testing phases. In UML-RT it is still possible to refine the model, but the refinements should be careful to preserve the properties that were successfully checked (the verification of the properties of component-based implementations in known run-time environments is the subject of an ongoing research activity).

As a final observation, it should be noted that UML-RT is also used for introducing components in UML-based development of non real-time software in a more effective way than using standard UML components in deployment diagrams. Our approach is not very effective for this kind of developments. In fact, although it is possible to build UML+ (and UML+-RT) models of *any* system, in general systems that do not exhibit a real-time behavior do not need to be translated and model-checked as discussed above.

6 Validation

The system described in Sect. 4.1 has to operate the crossing gate in a way that satisfies the following two properties:

Safety: The gate is closed during all occupancy intervals.

Utility: If no train is in any occupancy interval, nor within $\xi 1$ prior to an occupancy interval, nor within $\xi 2$ after an occupancy interval, then the gate is open.

Point X (see Fig. 4) is thus defined as follows: when a train enters zone X-II it is time to start closing the gate, so that the bar will be completely lowered when the train arrives at II. The exact position of X depends on the speed of the fastest trains. In order to have the gate closed when the fastest trains arrive at II, we must begin to close the gate dm-g time units after the train entered region R. If the train is slower the gate will be already closed when it enters the crossing region I.

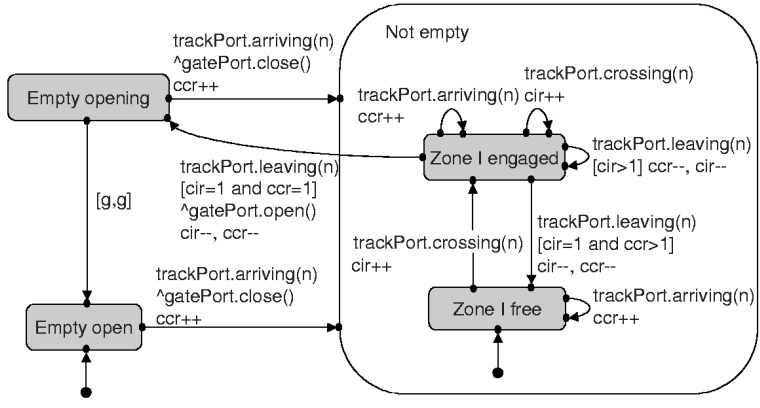


Fig. 12. The statechart of capsule controller

The behavior of the controller is defined as specified in Fig. 12. The idea is that the Controller counts the trains in the X-RO zone (by means of variable *ccr*) and in the II-

RO zone (by means of variable *cir*). As soon as *ccr* becomes grater than zero the controller sends the *close* command to the gate. When *ccr* and *cir* become zero the *open* command is sent to the gate. The signals *crossing* and *leaving* correspond to the signals generated by the sensors II and RO, while signal *arriving* corresponding to point X must be delayed of dm-g time units with respect to the signal generated by sensor RI. It is responsibility of the Track components (whose statecharts are omitted because of space reasons) to satisfy these rules. The parameter *n* of the *arriving, crossing* and *leaving* events is the identifier of the track which issues the signal.

An important observation: the statechart given in Fig. 12 cannot be analyzed by Kronos as it is, because integer variables (like *ccr* and *cir*) are not allowed in the timed automata analyzed by Kronos. Since the number of tracks is limited this problem can be easily solved by replacing the integer counters with finite state automata where each state represents a value. Moreover, the events cannot be parameterized: in the Kronos model *arriving(n)* must be replaced by *arriving1*, *arriving2*, …, *arrivingN* (where N is the number of tracks). This applies of course to *crossing* and *leaving* events as well.

The model of the GRC described above was translated into a set of timed automata that satisfies the required properties (this was proven by means of the Kronos model checker).

The model is relatively simple, thus it can be directly translated into a UML-RT model, but the latter would not always behave as the original UML+-RT model. In fact by applying our translator (which implements the concepts described in section 4) we obtain a set of warnings, indicating that a new model is needed to verify the actual behavior of the UML-RT system. The warnings by the translator concern the following issues:

- In the statechart of capsule Gate the reactions to events *open* and *close* are immediate. We need a queue to simulate non instantaneous transactions: in this case a 2 events, 2 place queue. The distance between an *open* and a *close* event is no less than g+hm (due to the behavior of the Controller). Being dt1 the time to handle one event, it must be 2 dt1 < (hm+g). For a real computer-based system this condition is very easily satisfied.
- In the statechart of capsules *CounterCir* and *CounterCcr* (not shown) there are instantaneous transactions. In the worst case, *N* fastest trains enter region *R* simultaneously: they will enter region X-II simultaneously, they will enter region II-RO simultaneously and finally they will exit region I simultaneously. We need 2 queues to manage these simultaneous transitions: one for *CounterCir* and one for *CounterCcr.* In both cases two pairs of events are relevant: *arriving-leaving* for *CounterCcr* and *crossing-leaving* for *CounterCir.* Both the queues are organized as follows (N being the number of tracks). The queues are a 2 events *2N* place queue. Being dt2 the time taken to process a single event, it must be (*2 N dt2*) < (*dm+hm*). In fact, in the worst case, *N* fastest trains entering simultaneously fill the queue with *2N* events, which must be handled in no more than (*dm+hm*) time units (i.e., before all the trains leave the crossing). For a real computer-based system this condition is very easily satisfied.
- Capsule Track generates signal *arriving* exactly *dm-g* time units after receiving the signal generated by sensor *RI*. Signal *arriving* is communicated instantly to

the Controller. This is unfeasible in a real-life environment. We have to give time to a real-world controller to react: for this purpose we specify that signal arriving is issued (*dm–g)–d_react* time units after the signal from the sensor is received, where *d_react* is the time taken by the real environment to react to an event (including event handling, transmission, etc.).

Following the principles illustrated in Sect. 5 we built a "more realistic" UML+-RT model. This model was then translated into UML-RT, and the resulting model was tested by means of AnyLogic, a tool that provides a simulation environment for UML-RT models. The simulation showed that the system actually behaves as required. The simulation can be seen at http://www.xjtek.com/applications/?area=traffic (the model can be downloaded from the same site). Of course the simulation does not guarantee that sooner or later an erroneous situation will occur. In order to exclude this possibility we need to formally prove the properties of the system. For this purpose, we also built –following the indications of the translator reported above– the model that reproduces the behavior of the environment. This model – translated into timed automata and checked by means of Kronos – showed that the system's behavior actually satisfies the requirements. Note that in order to satisfy the requirements, *d_react* has to be given a proper value. Such value can be computed taking into account the delays introduced by the queues and the characteristics of the model or – more simply – by means of a trial-and-error process. In fact by running Kronos with half a dozen different values of *d_react* we were able not only to find safe values, but also the minimum safe value.

The Kronos model was also used to find unsafe working conditions for the system (e.g., dm too short with respect to the speed of the gate). Modifying the simulation model accordingly we were able to simulate with AnyLogic the occurrence of erroneous situations.

7 Conclusions and Related Work

Several research activities were carried out in order to provide UML with formal semantics. However, such initiatives assume perfect technology (like [8]) or are oriented to providing UML with a precise underlying model (like [17]).

In fact, one of the main obstacles to the application of rigorous development techniques is the difference between real-time application software and functional design (which adopt simplifying assumptions, like instantaneous and perfect communications, synchrony of interaction with the environment, or atomicity of actions) and the physical real-time systems [19].

In order to solve the "synchrony assumption" Taxys provides a compiler that derives a verifiable model from programs written in Esterel and C [2]. In particular, the model includes the specification of an "event handler" that represents the interface between the external environment and the real-time application (and plays a role similar to our event queues).

The work presented here is an initial effort to tackle the problem of bridging a perfect, synchronous, specification-oriented UML-based formalism with the real, imperfect, asynchronous implementation world. In particular we addressed the conversion of UML+ (an extension of UML that is suitable for the specification of real-time systems [5]) into a notation suitable for implementation, namely UML-RT, with the constraint that the final model retains the properties of the original model. This goal required to add the concept of component (or capsule) to UML+, and to map the synchronous semantics of UML+ onto the asynchronous semantics of UML-RT. The first task was easily solved by borrowing the component notation of UML-RT. The graphical language obtained (called UML+-RT) integrates the component-oriented concepts of UML-RT with the elements of UML+, and specially with Timed Statecharts [11]. Much harder is the conversion of a synchronous model into an asynchronous one. We propose a translation process based on incremental refinements of the original UML+-RT model, guided by the translation tool and aiming at the complete removal of the features that model "ideal" situations that cannot be achieved in practice. In particular we propose to use two models, one to be translated into UML-RT for implementation, and another explicitly modeling the mechanisms of the real execution environment, in order to formally prove the properties of the system in real working conditions.

Figure 11 illustrates the development environment which exploits UML+-RT and the associated tools. The construction of the tools is complete, while their integration is in progress. The environment will enable a development process where the development is mainly based on model construction, and the relevant properties of the models can be checked step-by-step.

Acknowledgments. The work described here was carried out as part of the ITEA project DESS (Software Development Process for Real-Time Embedded Software Systems). Project DESS is partly supported by MIUR. More on DESS can be found at `http://www.dess-itea.org`.

Giuseppe Occorso was with CEFRIEL when he participated in the work reported here.

References

1. Alur, R., Dill, D.L.: A Theory of Timed Automata. Theoretical Computer Science, n.126 (1994) 183–235
2. Bertin, V., Closse, E., Poize, M., Pulou, J., Sifakis, J., Venier, P., Weil, D., Yovine, S.: Taxys = Esterel + Kronos. A tool for verifying real-time properties of embedded systems. Conference on Decision and Control, CDC'01. Orlando, IEEE Control Systems Society (2001)
3. del Bianco, V., Lavazza, L., Mauri, M.: An application of the DESS modeling approach: The Car Speed Regulator, In Proc. SIVOOES 2001, Budapest (2001)
4. del Bianco, V., Lavazza, L., Mauri, M.: A classification of real-time specifications complexity, In Proc. SIVOOES 2001, Budapest (2001)

5. del Bianco, V., Lavazza, L., Mauri, M.: An introduction to the DESS approach to the specification of real-time software. CEFRIEL Technical Report RT01002 (2001)
6. del Bianco, V., Lavazza, L., Mauri, M.: Model Checking UML Specifications of Real Time Software. The Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS02), Greenbelt (2002)
7. del Bianco, V., Lavazza, L., Mauri, M.: A Formalization of UML Statecharts for Real-Time Software Modeling. The 6th Biennial World Conference On Integrated Design Process Technology (IDPT 2002), "Towards a rigorous UML" session, Pasadena (2002)
8. Eshuis, R., Wieringa, R.: Requirements-Level Semantics For UML Statecharts. Formal Methods for Open Object-Based Distributed Systems – FMOODS'2000, Stanford (2000)
9. Ghezzi, C., Mandrioli, D., Morzenti A.: TRIO, a logic language for executable specifications of real-time systems. The Journal of Systems and Software, Vol. 12, n. 2. Elsevier Science (1990)
10. Heitmeyer C.L., Jeffords R.D., Labaw B.G., Comparing different approaches for Specifying and verifying Real-Time Systems, in Proc. 10th IEEE Workshop on Real-Time Operating Systems and Software, New York (1993) 122–129
11. Kesten, Y., Pnueli, A.: Timed and Hybrid Statecharts and their Textual Representation. In Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium (1992)
12. Lavazza, L., Quaroni, G., Venturelli, M.: Combining UML and formal notations for modelling real-time systems. In: Gruhn, V. (ed.): Proceedings of ESEC/FSE 2001, Vienna, ACM Press (2001)
13. Mauri, M.: Estensione di UML per la specifica e la verifica delle proprietà di sistemi Real Time. Politecnico di Milano, Thesis (in Italian), http://www.polimi.it (2001)
14. OMG: Unified Modeling Language Specification, Version 1.4. http://www.omg.org (2001)
15. Pnueli A., Shalev, M.: What is in a step: On the semantics of statecharts. In: Ito T., Meyer, A. R. (eds.): Intl. Conf. TACS '91: Theoretical Aspects of Computer Software. Lecture Notes in Computer Science, Vol. 526. Springer-Verlag, Berlin Heidelberg New York (1991) 244–264
16. Rational Software Corporation: Modeling Language Guide, Rational Rose® RealTime, `http://www.rational.com` (2002)
17. Reggio, G., Astesiano, E., Choppy C., Hussmann, H.: Analysing UML Active Classes and Associated State Machines–A Lightweight Formal Approach. In Maibaum, T. (ed.): Proc. FASE 2000– Fundamental Approaches to Software Engineering. Lecture Notes in Computer Science, Vol. 1783. Springer-Verlag, Berlin Heidelberg New York (2000)
18. Selic, B., Gullekson, G., Ward, P. T.: Real-Time Object-Oriented Modeling. Wiley (1994)
19. Sifakis, J.: Modeling real-time systems–challenges and work directions. EMSOFT01, Tahoe City, October 2001. Lecture Notes in Computer Science, Vol. 2211. Springer-Verlag, Berlin Heidelberg New York (2001)
20. Yovine, S.: Kronos, A Verification Tool for Real-Time Systems, Kronos User's Manual Release 2.2. Springer International Journal of Software Tools for Technology Transfer, Vol. 1 October (1997)

Modelling Recursive Calls with UML State Diagrams

Jennifer Tenzer and Perdita Stevens

Laboratory for Foundations of Computer Science
School of Informatics
University of Edinburgh
Fax: +44 131 667 7209
J.N.Tenzer@sms.ed.ac.uk, Perdita.Stevens@ed.ac.uk

Abstract. One of the principal uses of UML is the modelling of synchronous object-oriented software systems, in which the behaviour of each of several classes is modelled using a state diagram. UML permits a transition of the state diagram to show both the event which causes the transition (typically, the fact that the object receives a message) and the object's reaction (typically, the fact that the object sends a message). UML's semantics for state diagrams is "run to completion". We show that this can lead to anomalous behaviour, and in particular that it is not possible to model recursive calls, in which an object receives a second message whilst still in the process of reacting to the first. Drawing on both ongoing work by the UML2.0 submitters and recent theoretical work [1,6], we propose a solution to this problem using state diagrams in two complementary ways.

1 Introduction

The Unified Modelling Language [10] has been widely adopted as a standard language for modelling the design of (software) systems. One diagram type within UML is the state diagram, an object-oriented adaptation of Harel statecharts.

The use to which state diagrams are put varies with the type of project and the modeller's preferences, but a typical use is as follows. The modeller decides that some or all of the classes which are to appear in the system should be modelled with state diagrams; typically, classes which are perceived to have "interesting" state change behaviour will be so modelled, whereas those which are considered to be stateless or almost so will not be. For a given class, the modeller identifies the abstract states, which will be represented as *states* in the state diagram. This involves deciding which aspects of state are interesting, in that they may affect behaviour; it can be seen as choosing an equivalence relation on the set of (concrete, fully-detailed) possible states of objects of the class. Next, s/he considers which *events* may happen to an object of this class, and what effect those events have on the abstract state of the object.

In sequential single-threaded systems on which we concentrate in this paper a typical event is the receipt of a message which requests the synchronous invocation of an operation. The modeller may record that certain state transitions

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 135–149, 2003.

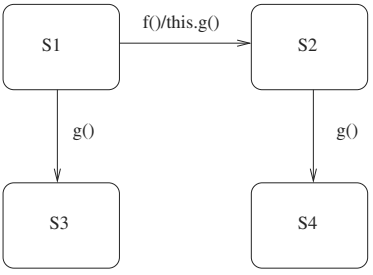


Fig. 1. Simple problem situation

happen only in particular circumstances; that is, s/he may add *guards* to the transitions. Finally, s/he may record how an object reacts to a given event happening in a given state by showing *actions* on the transitions (and/or within the states, but for simplicity we omit that possibility here). Typically, the modeller will not record every detail of the object's reaction, since that might involve placing the whole of an eventual method implementation as annotation on a transition in a diagram. A common compromise is to show only the messages which the object may *send* as part of its reaction to an event such as receiving a message, but not to show internal computation such as variable assignments.

Details vary, but essentially this process is recommended in many reputable sources, including for example Booch et al.'s *UML User Guide* [3] and the second author's own *Using UML* [14].

Unfortunately, this standard way of using UML is incoherent, given the semantics for UML state diagrams laid down in the UML standard. That is, following the UML semantics may yield nonsense for an apparently sensible collection of state diagrams. These are not pathological cases, either: there are common situations which cannot be modelled, with their intended semantics, using UML state diagrams as described above, and there are simple state diagrams, correct according to the UML standard, which cannot be interpreted. A very simple example is shown in Fig. 1. Suppose an object represented by this diagram is in state $S1$ and receives message $f()$, which causes the object to send itself the message $g()$.[1] What should the resulting state of the object be?

The UML semantics does not adequately cover cases like this. Probably the designer intended $S2$, with the idea that the implementation of $f()$ would involve extra actions besides the invocation of $g()$, but if we must consider the diagram as a complete machine, there is no good answer.

[1] More precisely, $f()$ is a *call event* and $g()$ a *call action* in the transition from $S1$ to $S2$. Call events are caused by call actions and are distinct from them [10] (3-148). In our example $g()$ on the transitions from $S1$ to $S3$ and $S2$ to $S4$ is a call event which is caused by the corresponding call action. Both call events and call actions are associated with an operation in UML, i.e. call events and actions in a state diagram model invocations of the operation of the object.

The fundamental problem seems to be that UML – like the UML community – is ambivalent about whether its state diagrams are intended to be *machines*, capable of being executed, or loose specifications, constraining a later implementation.[2] The UML semantics strongly suggests the former, but this does not always accord with how UML is used. In particular, when state diagrams are used to model synchronous message passing between objects in a sequential single-threaded system UML's run-to-completion semantics causes anomalies. Situations as shown above can be modelled by UML sequence diagrams and implemented in an object oriented programming language although the execution of corresponding UML state machines would result in a deadlock according to the default UML run-to-completion semantics (see example in Sect. 2.1). It is interesting to notice that Harel and Gery [7] were aware that recursive operation calls are problematic but apparently considered them unimportant. They wrote:

> ...when the client's statechart invokes another object's operation, its execution freezes in midtransition, and the thread of control is passed to the called object. Clearly, this might continue, with the called object calling others, and so on. However, a cycle of invocations leading back to the same object instance is illegal, and an attempt to execute it will abort.

We do not consider that the problem can be so easily dismissed. In object oriented design recursive calls occur frequently: for example, whenever any method is recursive, or when the Visitor or Observer pattern is used. In this paper we discuss the problem and propose a solution, drawing on both ongoing work by UML2.0 submitters and recent theoretical work [1,6].

The paper is structured as follows. The remainder of this section includes a note on standards and terminology. In Sect. 2 we explain the problem with the UML's current understanding of state diagrams. In Sect. 3 we introduce our solution, making use of two kinds of state diagrams. Section 4 formalises these diagrams and defines a suitable notion of consistency. In Sect. 5 we revisit the example introduced above; Sect. 6 discusses related work, and Sect. 7 concludes.

1.1 Note on Standards and Terminology

This paper is based on UML1.4, the current standard at the time of writing. It also draws on drafts of the new UML2.0 standard. The reader is assumed to be familiar with UML; it is important to note that the definition of UML is the OMG standard [10], not what is contained in any UML book.

In sequence diagrams, we show expressions on return arrows to indicate the value returned – this is common and harmless, but not actually described in [10].

[2] Even the terminology in the standard shows this tension: the terms state diagram, statechart diagram, statechart and state machine are not always used consistently (compare for example [10] (3-137) and (3-141)). In this paper we use "state diagram" as a general term, reserving "state machine" for an executable version.

We make several simplifying assumptions. We only consider sequential systems, so our state diagrams are assumed not to make use of concurrent substates. We treat rolenames identically with attributes; for example, our attribute environments include rolenames. Such an attribute has a class type, and its value will be an object identifier.

2 State Diagrams in UML

According to run-to-completion semantics, the action on a transition must have been completed before the transition is finished. If the action involves an object o of the class modelled by the state diagram making a call to another object p (and perhaps using the result of that call in some calculation), the action, and therefore the transition, does not complete until the call has been received by p, processed, and the result returned to o.

However, as soon as we consider the case that o and p might be the same object (recursion) or that part of p's reaction to the message from o might be to send o a new message (callback), it becomes clear that we cannot model situations with recursion or callbacks with UML state diagrams in which call events and actions involving calls are recorded on transitions. The next subsection demonstrates this using an example which we will use throughout the paper.

2.1 An Example of Callbacks with UML

The following example is used in different variants to illustrate several problems with callbacks in UML. The class diagram is given in the left part of Fig. 2.

Consider now an interaction between two objects as shown in the sequence diagram in the right part of Fig. 2. As response to an invocation of f object a calls method g of b and during the execution of g object b performs a callback to a. When the message setA2 arrives at a the execution of f is still in progress.

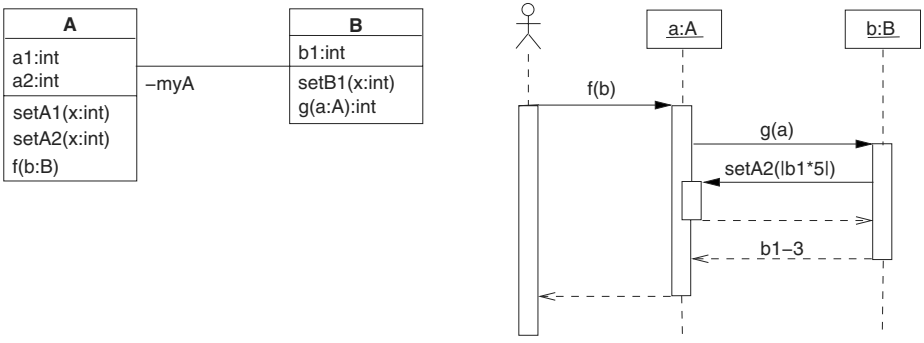


Fig. 2. Class diagram (left) and sequence diagram (right)

```
public void f(B b) {
    int y;
    y=b.g(this);
    if(a1*y>=0) {
        a1=y;
    }
}
```

```
public int g(A a) {
    myA=a;
    myA.setA2(Math.abs(b1*5));
    return(b1-3);
}
```

Fig. 3. Java implementation of methods f and g

An interaction like that can be implemented in Java without problems. One possible implementation is shown in Fig. 3.

The internal behaviour of objects of classes A and B can be modelled by state machines. In this example the state of an object depends on the signs of its attribute values. An object of A has four different states, an object of B two. The state machines in Fig. 4 show how methods affect the object states. An object of A is in one of the states in the top part of Fig. 4 if its value of a1 is negative, in one of the states at the bottom otherwise. Similarly the object is in one of the states on the left side of the diagram if a2 is negative, and on the right side if it is positive. Notice that these state machines intuitively correspond to the code in Fig. 3 under the assumption that internal computations are omitted in actions: the value of a2 is always positive after the completion of g due to the usage of the absolute value, and the sign of a1 is not changed in f, since the assignment of y to a1 is only carried out if y and a1 have the same sign.

For readability, an arrow may represent more than one transition with the same source and target states. The different transition labels are separated by commas. For example there are two transitions attached to the arrow from S1 to S2.

According to the UML semantics these state machines, considered together, do not behave in a sensible way. When a call of f arrives at an object a of A which is in state S1, the transition from S1 to S2 labelled by f is triggered, which invokes g. This causes a transition from S5 to itself and leads to a callback of setA2 to a, but a cannot react to this call because it is not in a stable state. The UML run-to-completion semantics prescribes that a can only process the call of setA2 after the transition from S1 to S2 has been completed, which will never happen.

Note that the same problem arises even if the callback is a query method, i.e. does not change any state.

3 Two Kinds of State Diagrams

We suggest handling the problem of callbacks by using two different kinds of state diagrams, one to model the overall effect of a method on the state of an object and the other to model the execution of actions of which this method consists. Thus we resolve the issue of whether state diagrams are loose specifications

or executable machines by providing both, for use in clearly defined different contexts.

3.1 Protocol State Machines (PSMs)

In UML1.4 [10] (2-170) PSMs are introduced as a state diagram variant, defined in the context of a classifier. We keep UML's terminology (PSM), but in our opinion these diagrams are best thought of as loose specifications, not as executable machines. They specify the permissible sequences of method calls on an object (the protocol), but not how the object will react to each method call. Their transitions (protocol transitions) are *allowed*, but not expected, to have action expressions. Here we only consider PSMs for classes and in order to enforce the separation between the diagram types, we follow a recent proposal for UML2.0 [11] where actions at protocol transitions are explicitly forbidden. The definition given below is a formalisation of a simplification of PSMs as presented in [11]. As notation for PSMs we use the standard UML state diagram notation as described in [10] (3.74.2). Thus a PSM is a simple form of UML state diagram, without actions: e.g., removing all actions from Fig. 4 yields PSMs. The designer would develop a PSM for classes that s/he considers to have interesting state change behaviour, as in current practice. The only difference is that instead of recording actions on the transitions, s/he will choose when it is worthwhile to record how reactions to events are implemented using another diagram type, as follows.

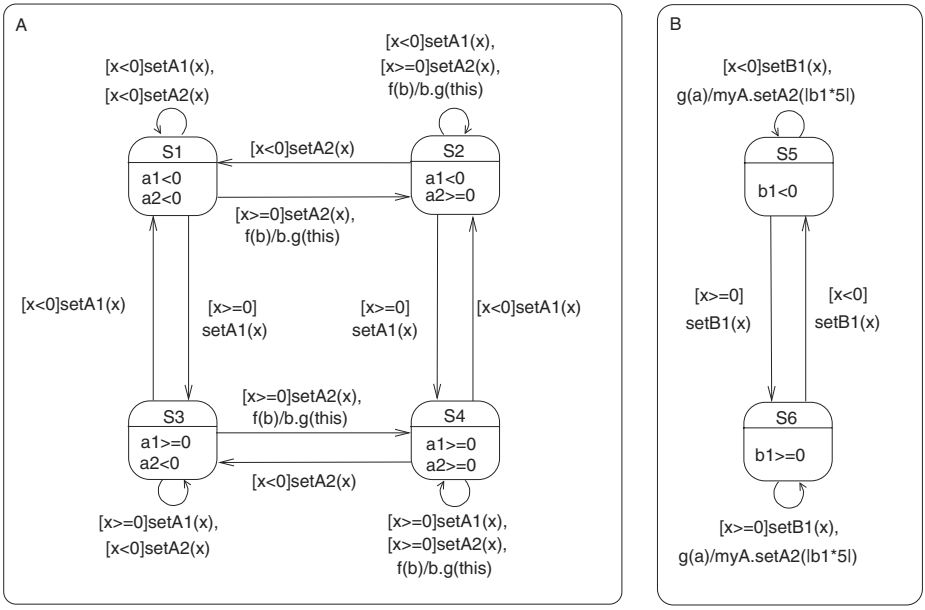


Fig. 4. State machines for A and B

3.2 Method State Machines (MSMs)

Both the current UML specification [10] and the proposal [11] allow the definition of a state diagram in the context of an operation, but do not provide detail about the particular features and behaviour of this kind of state diagram. We propose MSMs which are a simplified variant of *sequential class machines* as presented in [6], which in turn are a variant of *recursive state machines* as introduced in [1]; they allow recursion.

Figure 5 shows the three MSMs for f, setA2 and g which correspond to the Java implementation given in Fig. 3. Each MSM is represented by a box with rounded edges and is labelled by the class which owns the method, the method name and its parameter. The MSM for f consists of an *entry state* Fe, an *invocation box*, two *internal states* F1 and F2, and two *return states* Fr1 and Fr2. Most of the notation is standard UML. Entry states contain variable declarations and in the state diagram for g a return expression is shown within the return state Gr. Invocation boxes are shown as boxes with a double borderline and include a method call. Each box has an entry and an exit point, represented as shown. In the MSM for f a *return variable* is attached to the exit point of the invocation box. States and boxes are connected by transitions which are labelled by guards and actions, but not by events. In terms of the UML metamodel two kinds of events are relevant in MSMs: implicit completion events which cause normal transitions (as commonly used in UML activity diagrams), and return events, which permit the MSM to move on from a method invocation box. Since an MSM models how an activity is performed, it is bound to have similarities with an activity diagram. We add a precise semantics for MSMs in Sect. 4, especially, semantics for invocation of methods represented by other diagrams, which is not defined in UML activity diagrams. In this paper we will not discuss how our proposal sits inside the UML metamodel, since it would raise no interesting issues: instead, we focus on description and formalisation.

4 Formal Definitions and Consistency

Clearly, the designer will need to satisfy him/herself that the state diagrams, both PSMs and MSMs, contained in a given model are consistent: that is, that it is possible to implement methods according to the MSMs and have the resulting classes act in accordance with the PSMs. In this section we specify what this consistency means.

First we consider some informal examples:

- The MSMs in Fig. 5 are intuitively consistent with the PSMs obtained by deleting actions from Fig. 4; the transitions in the PSMs accurately reflect the state changes that occur when the MSMs are followed through in a natural way.
- However, if we change the invocation in the MSM for g from myA.setA2(|b1*5|) to myA.setA2(b1*5), we destroy consistency because if b1 is negative then a2 is set to a negative value. That means a call of f on

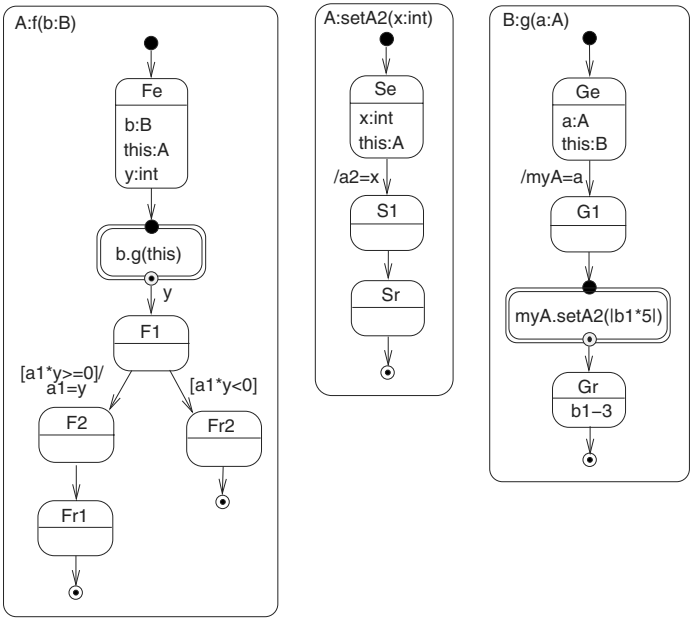


Fig. 5. Method state machines for f, setA2 and g

a does not always lead to an object configuration of a where a2 is positive, contradicting the PSM.

- Alternatively, if the MSM for f is altered so that a1 is always set to a1 * y (i.e. remove the guard on the transition from F1 to F2 and delete Fr2) we get a slightly more complicated inconsistency example, involving the states of two objects. If for instance a1 is positive and b1 is negative, then an invocation of f on a results in a configuration where a1 is negative. This is again a contradiction to the PSM for A which specifies that the sign of a1 is not supposed to change during the execution of f.

In order to formalise these intuitions we introduce formal definitions of both kinds of state diagrams. For purposes of exposition we use simplified forms of the diagrams; we believe, however, that most of the missing features of UML state diagrams could be added without serious problems. Our definitions of MSMs and their execution are adapted from definitions in [6] and [1].

We assume that there is a class diagram which defines a set of classes C. Each $c \in C$ is associated with a finite set $A_c = \{a_1 : T_1 \, ‹ \rhd\rhd\rhd ‹ \, a_n : T_n\}$ of typed attributes and a finite set M_c of methods, where each method $m \in M_c$ has a type $T_{cm} = cT_{cm} \times rT_{cm}$ defining the call and return type of the method[3].

Note that in the current work we are eliding the difference between the *operations* of classes and their *methods*. In UML these concepts are distinguished

[3] For simplicity we allow only one parameter and return value

in order to allow for inheritance: classes in an inheritance hierarchy may all have the same operation, inherited from a base class, which they implement using different methods. We do not consider inheritance here, since this raises many interesting questions about the appropriate inheritance of behaviour, and so the distinction between operation and method is unimportant. One possibility to explore would be that the protocol state machine would be written in terms of operations, and that the PSM defined for a base class would be inherited by subclasses. Then when a subclass provided its own method implementing an inherited operation, the designer could draw a new MSM for that method. This would open the door to considerations of behavioural subtyping: we could ask to what extent the MSMs for different methods implementing an operation were compatible.

A PSM is unsurprisingly simply a labelled transition system with guards:

Definition 1. *A protocol state machine (PSM) for a class c consists of*

- *a set S_c of states*
- *a labelled transition relation $\rightarrow \subseteq S_c \times L \times S_c$ where each label $l \in L$ is a tuple $(g, m(x))$ where $m \in M_c$ is a method name, x is a formal parameter of type cT_{cm}, and g is a Boolean expression over $A_c \cup \{x\}$ specifying the condition under which the transition may be taken. We do not prescribe the expression language used for g but we assume it can be evaluated to true or false given values for $A_c \cup \{x\}$.*

We will now define MSMs more formally. For a set $X = \{x_1 : T_1, \ldots, x_n : T_n\}$ of typed variables, a *variable environment* σ over X is a function $[x_1 \mapsto a_1, \ldots, x_n \mapsto a_n]$ where $a_i \in T_i \cup \bot_{T_i}$ for all i. The set of all variable environments over X is denoted by Σ_X. Attributes and attribute environments are treated in a similar way.

Moreover let $A = \bigcup_{c \in C} A_c$ be the set of all attributes and O the set of all object identifiers. An *object environment* is defined as a partial function $\Omega : C \rightarrow (O \rightarrow \Sigma_A)$ and the set of all object environments is denoted by Ω.

We do not prescribe an action language: we only specify that, given an object environment Ω and variable environment σ over X, an action is syntactically an expression over X, suitably extended with attribute selectors, for which an evaluation function $[\![ ]\!]_{\Omega\sigma}$ exists. We will later assume that the same evaluation function can be used to evaluate the guards used in PSMs. An action may not involve the invocation of methods or the creation or deletion of objects. Semantically an action α is a partial function $\alpha : (\Sigma_X \times \Omega) \rightarrow (\Sigma_X \times \Omega)$ expressing the effect the action has on the variable environment and object environment.

Definition 2. *A method state machine (MSM) for a method $m \in M_c$ consists of*

- *a set of local variables $X_{cm} = \{x_1 : T_1, \ldots, x_n : T_n\}$, including those mentioned below*

- *a set B_{cm} of invocation boxes as defined below*
- *a set of states S_{cm} partitioned into*
  - *a set I_{cm} of internal states*
  - *an entry state e_{cm} with formal parameters $x : cT_{cm}$ and* this $: c$ *in X_{cm}*
  - *A set of return states R_{cm} where each state $r \in R_{cm}$ has attached a return expression re over X_{cm} of type rT_{cm}*
  - *a set of box entry points $Entry_{cm}$ and a set of box exit points $Exit_{cm}$*
- *a transition relation $\rightarrow_{cm} \subseteq F \times Act \times T$ where $F = \{e_{cm}\} \cup I_{cm} \cup Exit_{cm}$, $T = R_{cm} \cup I_{cm} \cup Entry_{cm}$ and Act is a set of actions $\alpha : (\beta_{X_{cm}} \times \beta) \rightarrow (\beta_{X_{cm}} \times \beta)$*

Notice the "incompleteness" of the transition relation of an individual MSM: if the MSM reaches a box entry point, it cannot go further based on the definition of this MSM alone. This makes sense because we cannot know what the effect of the call on the environments should be. Later we will show how several MSMs interact to "complete" the transition relation.

Definition 3. *A method invocation box $b \in B_{cm}$ specifies*

- *an object expression oe, determining the target object*
- *a class identifier $d \in C$ determining the class*[4] *of oe*
- *a method identifier $m \in M_d$*
- *an argument expression $ae : cT_{dm}$*

A box is not itself considered to be a state in the MSM: instead it has two associated states:

- *an entry point $c_b \in Entry_{cm}$.*
- *an exit point $r_b \in Exit_{cm}$*

A return variable *$y : rT_{dm}$ from X_{cm} is defined to hold the value returned from the method invocation.*

Notice that any MSM is by definition well-typed, and that all method invocations occur in boxes.

4.1 Execution of MSMs

Suppose that we have a *closed set* $\mathcal{MSM}$ of MSMs: that is, each method invoked in an invocation box of an MSM in $\mathcal{MSM}$ is itself defined by an MSM in $\mathcal{MSM}$. We can then define the execution of MSMs in terms of a global state machine.

Let the sets of states and of boxes for each MSM be pairwise disjoint and let N be the set of all states, X the set of all variables, and B the set of all boxes. Before we specify a global state machine, we give definitions of a call stack and a global environment.

[4] As mentioned, we do not consider inheritance in this work, so polymorphism is not allowed: the class of the target object must be given statically.

Definition 4 (Call Stack). *A call stack $cs \in B^*N$ specifies the current position in each active MSM. It is a stack $b_1 : \triangleright\triangleright\triangleright : b_k : n$ of boxes b_i and a state n on top. It must satisfy a coherence condition as follows. Suppose that box b_j contains method identifier m_{b_j} and class identifier c_{b_j}. Then box b_{j+1} if $j < k$ (respectively the state n if $j = k$), must belong to the MSM for method m_{b_j} in class c_{b_j} (which must exist, by the assumption that we have a closed set of MSMs).*

Definition 5 (Global Environment). *Given a call stack $cs = b_1 : \triangleright\triangleright\triangleright : b_k : n$, a global environment $ge = \sigma_0 : \triangleright\triangleright\triangleright : \sigma_k \in \Sigma_X^*$ associated with cs is a stack of variable environments. It must satisfy a coherence condition as follows. For each $j \leq k$, σ_j is the local variable environment of the MSM containing box b_{j+1} if $j < k$, or of the MSM containing n otherwise.*

Definition 6 (Global State Machine). *A state of a global state machine (GSM) consists of a call stack $cs \in B^*N$, a global environment ge associated with cs, and an object environment σ.*

There are three kinds of transitions: as in a pushdown system, the applicable transitions are always determined by the state at the head of the stack. Suppose $cs = b_1 : \triangleright\triangleright\triangleright : b_k : n$ where n is a state in MSM, and let the global environment be $ge = \sigma_0 : \triangleright\triangleright\triangleright : \sigma_k$ and the object environment be σ.

1. *If n is an entry state, an internal state or a box exit state, the only possible transitions are* internal transitions *which are induced by transitions of MSM. Formally, suppose that $n \xrightarrow{\sigma} n'$ is a transition in MSM. Then $(b_1 : \triangleright\triangleright\triangleright : b_k : n \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma_k \frown \sigma) \rightarrow (b_1 : \triangleright\triangleright\triangleright : b_k : n' \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma'_k \frown \sigma')$ is a transition in the global state machine, provided that $\sigma(\sigma_k \frown \sigma) = (\sigma'_k \frown \sigma')$. (Note that if $(\sigma_k \frown \sigma)$ is not in the domain of the partial function σ, there is no transition.)*
2. *If n is a box entry state, the only possible transition is a* call transition, *pushing a new invocation onto the stack. If $n = c_{b_{k+1}}$, the entry state for a box which we now call b_{k+1}, let the object expression, class, method and argument expression specified in b_{k+1} be oe, c, m and ae respectively. Then let σ_{k+1} be a new variable environment over X_{cm} in which the formal parameter of m and* this *are bound to $[\![ae]\!]_{\sigma_k\sigma}$, $[\![oe]\!]_{\sigma_k\sigma}$ respectively. (If either evaluation fails, there is no transition). Let e_{cm} be the entry state of the MSM for method m in class c. Then $(b_1 : \triangleright\triangleright\triangleright : b_k : c_{b_{k+1}} \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma_k \frown \sigma) \rightarrow (b_1 : \triangleright\triangleright\triangleright : b_k : b_{k+1} : e_{cm} \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma_k : \sigma_{k+1} \frown \sigma)$ is a transition of the global state machine.*
3. *If n is a return state, the only possible transition is a* return transition, *popping the stack. If $n = r \in R_{cm}$, then $(b_1 : \triangleright\triangleright\triangleright : b_k : r \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma_{k-1} : \sigma_k \frown \sigma) \rightarrow (b_1 : \triangleright\triangleright\triangleright : b_{k-1} : r_{b_{k-1}} \frown \sigma_0 : \triangleright\triangleright\triangleright : \sigma'_{k-1} \frown \sigma)$ is a transition of the global state machine, where $r_{b_{k-1}}$ is the exit state for box b_{k-1}, and σ'_{k-1} is the environment σ_{k-1} updated by binding the return variable of box b_{k-1} to $[\![re]\!]_{\sigma_k\sigma}$ where re is the return expression associated with r (again, if re fails to evaluate, there is no transition).*

Note that the only non-determinacy in the global state machine is that arising from non-determinacy inside individual MSMs: if they are deterministic, so is the GSM. Notice also that the behaviour of the GSM respects the stack discipline. We will be most interested in how the GSM implements a particular method call. We write $s \overset{c\langle m}{\twoheadrightarrow} t$ when for some class c and method m, $s = (b_1 : \triangleright\triangleright\triangleright : b_k : e_{cm}\langle \square_0 : \triangleright\triangleright\triangleright : \square_k\langle \square)$, $t = (b_1 : \triangleright\triangleright\triangleright : b_k : r\langle \square_0 : \triangleright\triangleright\triangleright : \square'_k\langle \square')$ for some return state $r \in R_{cm}$ and there is some sequence of GSM transitions $s \rightarrow \triangleright\triangleright\triangleright t$, in which no intermediate state whose call stack contains at most k boxes has e_{cm} at the head of the call stack. Without the restriction on intermediate states we might inadvertently "catch" more than one invocation of m from within MSMs that have been activated earlier. Notice that if m is recursive the call stack grows each time m is invoked so that e_{cm} is allowed to appear as head of the call stack in this case.

4.2 Consistency between Protocol and Method State Machines

So far MSMs and PSMs are only connected by method names which are used to label transitions in PSMs and represent the context of a MSM. In this section we define what it means for a MSM to conform to a protocol which is specified by a PSM.

There can be no formal connection unless the designer has specified the precise meanings of the states in the PSM.[5] Accordingly, we assume that along with any PSM P for class c having states S_c we are given a function $h : \square_{A_c} \rightarrow S_c$ which maps an attribute environment to a state in P.

Definition 7 (Consistency). *Let G be a GSM defined by a closed set $\mathcal{MSM}$ of MSMs, and let P be a PSM for class c. G conforms to P with respect to a given initial global state gs if and only if whenever*

$$gs \rightarrow^* gs' \overset{c\langle m}{\twoheadrightarrow} gs''$$

(that is, a method is executed from some global state which is reachable from the initial state), where $gs' = (cs\langle \square_0 : \triangleright\triangleright\triangleright : \square\langle \square)$ and $gs'' = (cs'\langle \square_0 : \triangleright\triangleright\triangleright : \square'\langle \square')$ we have $h(\square(c)(\square(\mathsf{this}))) = s$ and $h(\square'(c)(\square'(\mathsf{this}))) = t$ where $s \overset{[g]m(x)}{\longrightarrow} t$ is a transition of P and $[\![g]\!]_{\square\square} = \mathsf{true}$.

Note that because the PSM plays no role in the execution of the global state machine, but acts as an independent specification of what it should achieve, it suffices to specify consistency with one PSM at a time. Note also that we are *not* requiring that every transition in the PSM has some counterpart in the GSM. This is deliberate: the PSM for a reusable class will specify all the capabilities of the class, not all of which may be used in a particular system (GSM).

An obvious question to ask is whether consistency is decidable. The answer depends on the choice of action language, but for any reasonable action language

[5] This is sometimes done in practice by adding constraints to the states of a state diagram.

Turing Machines can be coded as MSMs, in which case it is easy to reduce the Halting Problem to a consistency problem, which must therefore be undecidable. Nevertheless, some tool support is possible. For example, a tool might construct representative object configurations for the different combinations of abstract states in the PSMs, symbolically execute the MSMs and check against the PSM transitions. Even if the tool's checking was not exhaustive, it might find useful counterexamples, helping the user to develop the design.

5 Introductory Example Revisited

Finally we revisit the example discussed in the introduction, Fig. 1. If the designer modelled with PSMs and MSMs as introduced in this paper, the state diagram would be split into a PSM and MSMs for f and g. The PSM specifies unequivocally that an object in state S1 is in state S2 after f has been executed, whatever further invocations are performed during f.

Under the assumption that the set containing the MSMs for f and g is consistent with the PSM, an object calls g during the execution of the MSM for f when it is in an appropriate state, i.e. either in S1 or in S2. According to the specification of g in the PSM, the object is either in S3 or S4 after the execution of the MSM for g has finished, depending on which state it was in at the time of g's invocation. In either case the MSM for f has to perform further actions to guarantee that the object is in S2 after its completion, as specified in the PSM.

Variants of the notation might be considered. For example, we might permit annotation of transitions to show what callbacks were *expected* to happen. However, the designer of the class will not always be in a position to know what callbacks might happen, if other classes in the system are designed by other people. (This need not prevent the designer knowing what the state after the transition will be, given adequate contracts for called methods.)

6 Related Work

We have used work by others for our definitions of PSMs and MSMs. PSMs are mentioned in UML1.4 [10] and specified in more detail in U2 Partners' proposal for UML2.0 [11]. Since this proposal contains some remarks on inheritance, operations and methods are differentiated there.

The formalism for MSMs presented in this paper is based on recursive state machines, which have been defined in [1], and sequential class machines as introduced in [6]. Recursive state machines are extensions of ordinary state machines where a state can correspond to a possibly recursive invocation of a state machine. They can be used for modelling sequential imperative programs with recursive procedure calls. Besides a definition [1] also contains a complexity analysis of recursive state machines concentrating on reachability and cycle detection. Similar results have been achieved independently in [2].

Class machines are an object oriented extension of recursive state machines. The semantic definition of their sequential variant in [6] covers exceptions, inheritance and object creation in addition to what we have presented as MSMs. Adding a mechanism for multi-threading leads to the definition of concurrent class machines. In contrast to our work class machines are considered in isolation, not in conjunction with a more abstract modelling technique.

There is much work on formalisation of UML state diagrams; we only present a small subset. All of the approaches differ from ours in that they do not consider the problem of recursive calls. In [12] a formalisation with labelled transition systems and algebraic specifications written in the specification language CASL is presented. Labelled transition systems are also suggested as formalism in [13], where a structured operational semantics for UML state diagrams is introduced. Both in [5] and in [8] graph transformations are used as basis for state diagram formalisation, but these works differ in detail; [4] uses ASMs. In [9] state diagrams are first mapped to extended hierarchical automata and then a semantics for these specific automata is defined in terms of Kripke structures.

7 Conclusions and Further Work

We have pointed out that the current UML semantics for state diagrams is not sensible for situations involving recursive method calls. After showing this problem on an example we have presented an alternative approach for modelling the internal behaviour of objects using UML. In contrast to the current version of UML we differentiate between a loose specification of the effect of a method on an object and an executable machine representing an implementation of a method. We have introduced PSMs and MSMs for these purposes and defined what it means for a set of MSMs to be consistent with a PSM.

In future we would like to consider tool support; indeed we undertook this work because the recursive call problem prevented us from making progress with work on providing tool support for the concurrent development of state and sequence diagrams. For practical use, more complex MSMs would be needed, allowing for object creation for example. Most of that work would be routine; the exception would be adding inheritance, which as briefly mentioned in Sect. 4 would raise both theoretical questions and issues in practical modelling.

Acknowledgements. We are grateful to the British Engineering and Physical Sciences Research Council for funding (GR/N13999/01, GR/A01756/01), and to the referees for helpful comments on presentation.

References

[1] R. Alur, K. Etessami, and M. Yannakakis. Analysis of recursive state machines. In *Computer Aided Verification*, pages 207–220, 2001.

[2] M. Benedikt, P. Godefroid, and T.W. Reps. Model checking of unrestricted hierarchical state machines. In *Automata, Languages and Programming*, pages 652–666, 2001.

[3] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley Longman, 1998.

[4] 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.

[5] M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In Manfred Broy, Derek Coleman, Tom S. E. Maibaum, and Bernhard Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.

[6] R. Grosu, Yanhon A. Liu, S.A. Smolka, S.D. Stoller, and J.Yan. Automated software engineering using concurrent class machines. In *Proc. of the 16th IEEE International Conference on Automated Software Engineering, ASE'01*. IEEE, 2001.

[7] D. Harel and E. Gery. Executable object modeling with statecharts. *IEEE Computer*, 30:7:31–42, 1997.

[8] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 241–256, 2001.

[9] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In *Proc. FMOODS'99, IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems, Florence, Italy, February 15-18, 1999*. Kluwer, 1999.

[10] 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`.

[11] U2 Partners. Unified Modeling Language 2.0 proposal, version 2 beta R1, September 2002.

[12] G. Reggio, E. Astesiano, C. Choppy, and H. Hussmann. Analysing UML active classes and associated state machines – A lightweight formal approach. In *FASE 2000 – Fundamental Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*, pages 127–146, 2000.

[13] M. von der Beeck. Formalization of UML-Statecharts. In Martin Gogolla and Cris Kobryn, editors, *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools*, volume 2185 of *Lecture Notes in Computer Science*, pages 406–421, 2001.

[14] Perdita Stevens with Rob Pooley. *Using UML: software engineering with objects and components*. Addison Wesley Longman, 1999 (updated edition).

Pipa: A Behavioral Interface Specification Language for AspectJ*

Jianjun Zhao[1] and Martin Rinard[2]

[1] Department of Computer Science, Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp
[2] Laboratory for Computer Science, Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139, USA
rinard@cag.lcs.mit.edu

Abstract. Pipa is a behavioral interface specification language (BISL) tailored to AspectJ, an aspect-oriented programming language. Pipa is a simple and practical extension to the Java Modeling Language (JML), a BISL for Java. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. This paper discusses the goals and overall approach of Pipa. It also provides several examples of Pipa specifications and discusses how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification. The goal is to facilitate the use of existing JML-based tools to verify AspectJ programs.

1 Introduction

Aspect-oriented programming (AOP) has been proposed as a technique for improving the separation of concerns in software design and implementation [2,13, 17,19]. AOP provides explicit mechanisms for capturing the structure of crosscutting aspects of the computation such as exception handling, synchronization, performance optimizations, and resource sharing. Because these aspects crosscut the dominant problem decomposition, they are usually difficult to express cleanly using standard languages and structuring techniques. AOP can eliminate the code tangling often associated with the use of such standard languages and techniques, making the program easier to develop, maintain, and evolve.

The field of AOP has, so far, focused primarily on problem analysis, language design, and implementation. The specification and validation of aspect-oriented programs has received comparatively little attention.

To formally verify aspect-oriented programs, we must have some means to formally specify the properties of aspect-oriented programs. Note that, because

* This work was carried out during Jianjun Zhao's visit to Laboratory for Computer Science, Massachusetts Institute of Technology.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 150–165, 2003.

AOP introduces new concepts such as join points, advice, instruction, and aspects, existing formal specification languages can not be directly applied to AOP languages. This motivates us to design a formal specification language that is appropriate for specifying programs written in AOP languages.

Instead of designing a generic specification language for AOP, we choose instead to design a *behavioral interface specification language* (BISL) tailored to AspectJ, an aspect-oriented extension to Java [14]. A *behavioral interface specification* describes both the details of a module's interface with clients and its behavior from the client's point of view [16]. By using a BISL, we are able to formally specify both the behavior and the exact interface of AspectJ programs' modules, which is an essential step towards the formal verification of these modules.

Our BISL for AspectJ is called Pipa, which is a simple and practical extension to Java Modeling Language (JML) [16], a widely accepted BISL for Java. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. Pipa provides annotations to specify AspectJ programs with pre- and postconditions, class invariants, and aspect invariants. These annotations enable both dynamic analysis in support of activities such as debugging and testing and static analysis in support of the formal verification of properties of AspectJ programs. Static analysis activities could verify that the code of advice of an aspect correctly implements its specification, the specification of advice in an aspect is compatible with the specification of the method in a class that the advice advises, and the correctness of the aspect weaving process.

The key to the verification process is to develop a transformation tool that automatically transforms an AspectJ program, together with its Pipa specification, into a corresponding Java program and JML specification. By doing so, some JML-based checking and verification tools [16,6,10] can be used directly to check and verify AspectJ programs.

In this paper, we discuss the goals of Pipa and its overall specification approach. We also provide examples of how to use Pipa to specify AspectJ aspects, and discuss how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification, which is a crucial step towards the utilization of existing JML-based tools to verify AspectJ programs.

The rest of the paper is organized as follows. Section+2 presents the design rationale for Pipa. Section 3 briefly introduces the Java Modeling Language. Section 4 uses some examples to show how AspectJ aspects are specified in Pipa. Section 5 discusses the issues about specification inheritance and crosscutting. Section 6 discusses how to transform an AspectJ program together with its Pipa specification into a standard Java program and JML specification. Section 7 discusses related work; we conclude in Sect. 8.

2 Design Rationale

Our purpose is to understand how to formally specify and verify aspect-oriented programs. Several questions focus our investigation. First, what is an aspect invariant, and how would we specify it? How would one specify aspects that contain around advice which may alter the return value of a method in the base code? What is the contract checking semantics between an aspect and the base code? How would one verify that the behavior of an aspect does not violate the desired functionality of the base code? How would one verify that aspects whose advice affects the behavior of base code still provide an acceptable semantics according to the specification of the base code? How would one verify the correctness of a woven program after weaving the aspect and base code?

Designing Pipa is therefore only a part of our proposed activities. We must also develop techniques and tools to support the formal specification and verification of AspectJ programs augmented with Pipa specifications. We have therefore chosen to design Pipa as a compatible extension to JML to (1) facilitate its adoption by current JML users, and (2) facilitate the adoption of existing JML-based tools to check AspectJ programs. JML is an especially appropriate base for the Pipa design for two reasons. First, AspectJ is a seamless extension to Java for implementing crosscutting concerns, and JML is a BISL specially designed for Java. By structuring Pipa as an extension to JML, we can focus our attention on the new issues associated with the use of aspects. Second, (JML has efficient tool support for both static and dynamic checking of Java programs) if we can transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification, we can use JML-based tools directly to verify AspectJ programs. Based on these considerations, we keep the following issues in mind to make Pipa as compatible with JML as possible.

- Each legal JML specification for Java should also be a legal Pipa specification for AspectJ.
- Specifying AspectJ programs with Pipa should feel like a natural extension of specifying Java programs with JML.
- It should be possible to extend existing JML-based tools (such as static checking tools, run-time assertion checking tools, documentation tools, and design tools) to support Pipa in a natural way.
- The Pipa specifications themselves should be strongly connected to the AspectJ code, as JML specifications are connected to Java code.

Like JML, Pipa specifications are also expressed as Javadoc-style comments in AspectJ interface definitions, enclosed between `/**` and `*/`. Pipa therefore permits the specification to be embedded in regular AspectJ files.

To focus on the key ideas of Pipa, in this paper we do not consider the specification of AspectJ classes and interfaces. These classes and interfaces can be specified in Pipa in a similar way as JML [16]. Moreover, we only consider join points related to method and constructor calls, and introductions that introduce members such as methods and constructors.

3 The Java Modeling Language

The Java Modeling Language (JML) [16] is a formal BISL tailored to the Java programming language [7]. JML allows assertions (pre- and postconditions and class invariants) to be specified for Java classes and interfaces. JML adopted the model-based approach of Larch [8] by supporting specification-only model fields. These fields describe abstractly the value of objects and are used only for specification purposes. The predicates in JML are written using regular Java expressions extended with logical operators and universal and existential quantifiers.

JML specifications are expressed as special comments in Java interface definitions, following `//@` or enclosed between `/*@` and `*/`. JML also permits the specification to be embedded in regular Java files. In addition, JML specifications can also be expressed as Javadoc-style comments, that is, enclosed between `/**` and `*/`.

JML supports specifying a class at both the method and the class level. These two specifications together form the complete behavioral specification for the class. For example, the following shows a simple method-level specification:

```
/**@  public normal_behavior
   @    requires x >= MIN_X && x <= MAX_X;
   @    ensures true;   */
public void moveX(int x) { ...  }
```

The specification states that if a precondition (represented by a `requires` clause) `x >= MIN_X && x <= MAX_X` holds at the call of a public method, the method terminates normally, i.e., does not throw an exception, and the postcondition (represented by an `ensures` clause) `true` is satisfied at the end of the method call. There is a variation for the `normal_behavior` specification, i.e., a `behavior` specification, which can be used to specify the conditions under which a method may, may not, or must throw an exception.

JML also provides class-level specifications with additional clauses such as `invariant`, `constraint`, and `model`. A `model` clause allows the declaration of so-called model variables which are variables that exist only within a specification. Such variables are often used to refer to the internal state of an object. An `invariant` clause in JML declares those properties that are true in all publicly visible, reachable states of an object, i.e., for each state that is outside of a public method's execution. An invariant is supposed to be established by the class constructors and to be preserved by each (public) method. A `constraint` clause is used to specify how values may change between earlier and later states, such as a method's pre-state and its post-state.

In addition, a specification in JML may be composed of several cases separated by the keyword `also`; which states that when the precondition of one case holds, the rest of that case's specification must be satisfied. JML also supports specification inheritance. A subtype inherits the specifications from its supertype's public and protected members (i.e., fields and methods), as well as its invariants and history constraints as additional specification cases.

4 Aspect Specifications

In AspectJ, an aspect is a modular unit for implementing crosscutting concerns. Its definition is similar to a Java class, and can contain methods, fields, and initializers. Pointcuts and advice support the implementation of crosscutting aspects. Pipa can specify an aspect at both the module and the aspect level. Pipa uses *module-level specifications* to specify the behavior of individual modules such as advice, introduction, and methods in an aspect, and *aspect-level specifications* to specify the global properties of the aspect as a whole. In this section, we show how advice and introduction can be specified using Pipa; method specification in Pipa is the same as in JML. Through this paper, we introduce the specification approach of Pipa with the use of an example AspectJ program (taken from [1] with slight modifications), which consists of two classes `Point` and `Line` (both shown in Fig. 1) and several aspects.

4.1 Advice Specifications

Advice defines pieces of code in an aspect that should be executed when a pointcut is reached during the execution of a program. AspectJ provides three kinds of advice, that is, *before*, *after*, and *around* advice [1]. The most significant feature of advice is that it can dynamically change the behavior of a class it advises, and therefore implements *behavioral crosscutting*.

In Pipa, the specification of a piece of advice is similar to that of a method in JML: the advice is annotated with preconditions, postconditions, and frame conditions; these are declared, respectively, with `requires`, `ensures`, and `modifies`

```
class Point {
  int x, y;
  /**@  model instance int x_Mdl, y_Mdl; */

  /**@  depends x_Mdl <- x; */
  /**@  represents x_Mdl <- x; */

  /**@  depends y_Mdl <- y; */
  /**@  represents y_Mdl <- y; */

  /**@  public behavior
     @    assignable x_Mdl;
     @    ensures x_Mdl == x;
     @    signals (Exception z) false; */
  public void setX(int x) {
     this.x = x;
  }
  /**@  public behavior
     @    assignable y_Mdl;
     @    ensures y_Mdl == y;
     @    signals (Exception z) false; */
  public void setY(int y) {
     this.y = y;
  }
}
```

```
class Line {
  private Point p1, p2;
  /**@  model instance Point p1_Mdl, p2_Mdl; */

  /**@  private depends p1_Mdl <- p1; */
  /**@  private represents p1_Mdl <- p1; */

  /**@  private depends p2_Mdl <- p2; */
  /**@  private represents p2_Mdl <- p2; */

  /**@  public behavior
     @    assignable p1_Mdl;
     @    ensures p1_Mdl == p1;
     @    signals (Exception z) false; */
  public void setP1(Point p1) {
     this.p1 = p1;
  }
  /**@  public behavior
     @    assignable p2_Mdl;
     @    ensures p2_Mdl == p2;
     @    signals (Exception z) false; */
  public void setP2(Point p2) {
     this.p2 = p2;
  }
}
```

Fig. 1. Two classes `Point` and `Line` with their Pipa specifications

```
aspect PointBoundsPreCondition {
 /**@  public behavior
    @    requires x >= MIN_X && x <= MAX_X;
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires x < MIN_X || x > MAX_X;
    @    ensures false;
    @    signals (Exception z)
    @      z instanceof RuntimeException; */
 before(int x): call(void Point.setX(int))
                  && args(x) {
     if ( x < MIN_X || x > MAX_X )
        throw new RuntimeException();
 }
}
```

(a)

```
aspect PointBoundsPostCondition {
 /**@  public behavior
    @    requires p.getX() == x;
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires p.getX() != x;
    @    ensures false;
    @    signals (Exception z)
    @      z instanceof RuntimeException; */
 after(Point p, int x): call(void Point.setX(int))
                          && target(p) && args(x) {
    if ( p.getX() != x )
       throw new RuntimeException();
 }
}
```

(b)

Fig. 2. (a) A piece of before advice with its Pipa specification. (b) A piece of after advice with its Pipa specification

clauses. These preconditions, postconditions, and frame conditions together form the *specification* of the advice, which can be used to verify the code of the advice.

In contrast to the pre- and postcondition of a method, which are associated with method call and return, the pre- and postcondition of advice is defined in terms of the principle of control flow transferring [5]. This is because each piece of advice in AspectJ is automatically woven into the method(s) it advises by a compiler (called `ajc`) during the aspect weaving process. Advice therefore executes in response to certain program actions, instead of being directly invoked like a method. From this viewpoint, the pre- and postcondition of a piece of advice can be defined as follows:

- A *precondition* for a piece of advice is an assertion that states the properties that must hold before the control flow is transferred into the advice.
- A *postcondition* for a piece of advice is an assertion that states the properties that the advice must establish before the control flow returns to the advised method.

Before Advice. Before advice executes when a join point is reached and before the computation proceeds [1]. It may execute when computation reaches the method call and before the actual method starts executing. In Pipa, the behavior of before advice can be specified by a precondition, a postcondition, and a frame condition.

As an example, consider the aspect `PointBoundsPreCondition` shown in Fig. 2a, which declares a piece of before advice that modifies the behavior of the class `Point`'s `setX` method. The advice can be applied to each join point where a target object of type `Point` receives a method call with signature `void Point.setX(int)`. The `args` keyword denotes the argument of the method call.

The specification of this before advice is associated with the advice code, which contains two specification cases combined by a keyword `also`. For the first case, a `requires` clause supplies a normal precondition, which must be satisfied before control transfers to the advice from the method `setX`, and an `ensures` clause provides a normal postcondition, which must hold before control transfers to the advised method `setX`. The specification states that if the advice is entered with `x >= MIN_X` and `x <= MAX_X`, control must flow to the advised method `setX`. The implicit frame condition for this case means that no relevant locations may be assigned when this precondition holds. The second specification case states that if the advice is entered with `x < MIN_X` or `x > MAX_X`, the control must return to the caller of the method `setX` by throwing a `RuntimeException`.

After Advice. After advice executes after the computation under the join point terminates [1]. It may execute after the method body has executed, and just before control is returned to the caller of the method. AspectJ supports three kinds of after advice, that is, *after*, *after-returning*, and *after-throwing* advice. After-returning advice executes just after each join point, but only when the advised method returns normally. After-throwing advice executes just after each join point, but only when the advised method throws an exception of type `Exception`. after advice executes just after each join point, regardless of whether the advised method returns normally or throws an exception. Like before advice specifications, after advice specifications have a precondition, a postcondition, and a frame condition.

As an example, consider the aspect `PointBoundsPostCondition` shown in Fig. 2b, which declares a piece of normal after advice that modifies the behavior of class `Point`'s `setX` method. The after advice can be applied to each join point where a target object of type `Point` receives a method call signature `void Point.setX(int)`. The `target` and `args` keywords denote the target object and the argument of the method call.

The Pipa specification for the after advice has two specification cases. The first case states that if the advice is entered with `p.getX() == x`, the control must flow to the advised method. The second case states that if the advice is entered with `p.getX() != x`, control must return to the caller of the method `setX` by throwing a `RuntimeException`.

Around Advice. Around advice executes when a join point is reached, and has explicit control over whether the computation under the join point is allowed to execute at all [1]. around advice differs from before advice and after advice in the sense that it may execute code both before and after the method, and by (optionally) executing the `proceed` call, which causes the original method under the join point to execute. To specify around advice, Pipa borrows some ideas from [5]. Pipa uses the `proceeds` *predicate* clause, taken from [5], to state that when control flow proceeds to the original method body (or to any additional advice if present), the around advice must make *predicate* hold. Pipa also uses a keyword `then`, also taken from [5], to divide the specification of the around

```
aspect PointBoundsEnforcement {
 /**@  public behavior
    @    requires x >= MIN_X && x <= MAX_X;
    @    proceeds true;
    @    signals (Exception z) false;
    @  then
    @    ensures true;
    @    signals (Exception z) false;
    @ also
    @  public behavior
    @    requires x < MIN_X;
    @    proceeds false
    @    signals (Exception z) false;
    @  then
    @    assignable p.x_Mdl;
    @    ensures p.x_Mdl == MIN_X;
    @    signals (Exception z) false
    @ also
    @  public behavior
    @    requires x > MAX_X;
    @    proceeds false;
    @    signals (Exception z) false;
    @  then
    @    assignable p.x_Mdl;
    @    ensures p.x_Mdl == MAX_X;
    @    signals (Exception z) false  */
  void around(Point p, int x):
         call(void Point.setX(int))
                && target(p) && args(x) {
    if (x >= MIN_X && x <= MAX_X ) {
        proceed(p, x);
    } else if (x < MIN_X) {
        p.setX(MIN_X);
    } else {
        p.setX(MAX_X);
    }
}
```

Fig. 3. A piece of around advice with its Pipa specification

advice into two parts: the *before part*, which is the portion of the specification corresponding to the around advice before proceeding to the original method, and the *after part*, which is the portion corresponding to the around advice after returning from the original method but before returning to the original caller. With these specification constructs, Pipa can specify around advice conveniently.

As an example, consider the aspect `PointBoundsEnforcement` shown in Fig. 3, which declares a piece of around advice to modify the behavior of class `Point`'s `setX` method. The around advice can be applied to each join point where a target object of type `Point` receives a call to its method with signature `void Point.setX(int)`. The `target` and `args` keywords are used to assign the name to the target object and the argument of the method call. The code of around advice states that if `x` is greater than `MIN_X` and less than `MAX_X`, then the statement `proceed(p, x);` causes control flow to transfer to the original `setX` method body with the same arguments as the original invocation. Otherwise, in the `else` clauses the advice calls the `setX` method on `Point` directly.

The specification of the around advice, which is associated with the advice code, has three specification cases combined using the keyword `also`. The first case applies when `x >= MIN_X` and `x <= MAX_X`. The `proceeds true` clause states that control flow can always proceed to the original method (`setX`) body. The part following the `then` keyword states that after control flow transfers to the original method (`setX`) body, it then returns to the original client.

The second case, following the first `also` keyword, concerns the case where `x < MIN_X`. The `proceeds false` clause states that the control never proceeds to the original method (`setX`) body. The part following the `then` keyword also has a `requires` clause as a postcondition of the case. The `assignable` and `ensures` clauses state that control may return to the original client with possible mutation to `p`'s `x_Mdl` model field and with the given postcondition predicate

satisfied. The third case, following the second `also` keyword, which concerns the case where `x > MAX_X`, can be explained similarly.

4.2 Introduction Specifications

While a piece of advice can change the behavior of the classes it crosscuts, it can not change the static type structure of the classes. AspectJ provides a form called *introduction* that can operate over the static structure of type hierarchies [1]. An aspect can use introduction to add new fields, constructors, or methods into given classes or interfaces.

Pipa's introduction specification mechanism is similar to its method specification mechanism. Each piece of introduction may be annotated with preconditions (declared by `requires`), postconditions (declared by `ensures`), and frame conditions (declared by `modifies`). These preconditions, postconditions and frame conditions together form the *specification* of the introduction, which can be used to verify the code of the introduction.

As an example, consider the aspect `Introduction` shown in Fig. 4, which uses the `new` keyword to introduce a constructor into the `Point` and `Line` classes. These two constructors have two `int` parameters and the same body. The Pipa specification for the introduction is tailored to the code and states that if the introduction is called with `x >= 0`, the call must return to the caller normally.

4.3 Aspect Invariants

The previously discussed pre- and postconditions specify properties of individual modules such as advice, introduction, and aspect methods. There is also a need, however, to express global semantic or integrity properties for the aspect as a whole. Aspect invariants express these kinds of properties. Such invariants may involve only attributes, attributes and modules, or different modules in an aspect. Informally, an invariant of an aspect is a set of assertions (i.e., invariant clauses) that each instance of the aspect will satisfy at all times when the state is observable. Pipa uses an `invariant` clause, borrowed from JML, to specify aspect invariants.

As an example, consider the `Buffering` aspect shown in Fig. 5, which implements a buffering function. The first aspect invariant states that the value of the

```
aspect Introduction {
   /**@ public behavior
      @   assignable x_Mdl;
      @   requires x >= 0;
      @   ensures x_Mdl == x;
      @   signal (Exception z) false;   */
   public int (Point || Line).new(int x) {
        this.x = x;
   }
}
```

Fig. 4. A piece of introduction with its Pipa specification

```
public aspect Buffering pertarget(target(FileOutputStream)) {
  /**@ public invariant counter >= 0;
     @ public invariant buff =! null && buff.length == BUFF_SIZE; */
  private static final int BUFF_SIZE = 256;
  int counter = 0;
  byte[] buff = new byte[BUFF_SIZE];

  pointcut writeByte(byte[] bytes):
  void around(byte[] bytes): throws IOException: writeBytes(bytes) { }
}
```

Fig. 5. An aspect `Buffering` with aspect invariants

aspect's field `counter` must be greater than or equal to zero. The second states that the aspect's field `buff` is not null and the array it refers to has exactly `BUFF_SIZE` (256) elements.

5 Aspect Specification Inheritance and Crosscutting

We next discuss aspect specification inheritance and crosscutting in Pipa.

5.1 Specification Inheritance

The inheritance rules in AspectJ are (1) an aspect can only extend an abstract aspect; it can not extend a concrete aspect, (2) an aspect can extend a class, and (3) an aspect can implement any number of interfaces. According to these inheritance rules, we design some specification inheritance rules for Pipa as follows.

- A subaspect inherits the specifications of its superaspect's public and protected members (fields, methods, advice, introductions, and pointcuts), as well as the public and protected aspect invariants.
- A subaspect inherits the specifications of its superclass's public and protected members (fields and methods), as well as the public and protected class invariants.
- A subaspect inherits the specifications of its superinterface's public and protected members (fields and methods), as well as the public and protected interface invariants.

These aspect inheritances can be thought of as textually copying the public and protected specifications of the advice, introduction, or methods of an aspect's superaspects and superclasses and all interfaces that an aspect implements into the aspect's specification and combining the specifications using `also` keyword. By the semantics of advice, introduction, or method combining using `also`, in addition to any explicitly specified behaviors, these behaviors must all be satisfied by the advice, introduction, or method.

5.2 Specification Crosscutting

AspectJ supports both structural crosscutting (by means of introduction) and behavioral crosscutting (by means of advice) to modify the type structure and the behavior of classes an aspect crosscuts. Pipa should also be able to specify these structural and behavioral crosscutting issues at specification level. To make these possible, in the following we design some specification crosscutting rules for Pipa.

- The specification of an aspect's advice crosscuts the specifications of those classes' methods that the advice crosscuts (*behavioral crosscutting*).
- The specification of an aspect's introduction crosscuts the specifications of those classes that the introduction crosscuts (*structural crosscutting*).

These crosscutting rules ensure that an aspect specifies the structural and behavioral crosscutting of the one or more classes it crosscuts. This crosscutting can be thought of as syntactically (1) weaving the specification of a piece of advice in an aspect into the specification of each advised method in a class or different classes, and (2) weaving the specification of a piece of introduction in an aspect into the specification of one or more classes augmented by the introduction.

As an example of behavioral crosscutting, consider the aspect `DisplayUpdating` shown in Fig. 6, which modifies the behavior of methods in classes `Point` and `Line` shown in Fig. 1. `DisplayUpdating` declares a piece of after advice that can be applied to each join point where a target object of type `Point` receives a call to the method with signature either `void Point.setX(int)` or `void Point.setY(int)`. Also this after advice can be applied to each join point where a target object of type `Line` receives a call to the method with either signature `void Line.setP1(Point)` or signature `void Line.setP2(Point)`. In this case, the after advice in `DisplayUpdating` can affect the behavior of these methods in `Point` and `Line`. The specification of the after advice therefore should crosscut the classes `Point` and `Line`. This crosscutting can be thought of as syntactically weaving the specification of the after advice in `DisplayUpdating` into the specification of each advised method `setX`, `setY`, `setP1`, or `setP2`.

As an example of structural crosscutting, consider the aspect `Introduction` shown in Fig. 4, which publicly introduces two methods, one in class `Point`

```
aspect DisplayUpdating {
  pointcut move():
    call(void Line.setP1(Point)) ||  call(void Line.setP2(Point)) ||
    call(void Point.setX(int))   ||  call(void Point.setY(int))

  after(): move() {
    Display.update();
  }
}
```

Fig. 6. A piece of after advice that crosscuts two classes `Point` and `Line`

and another in class `Line`. According to the specification crosscutting, the Pipa specification for the introduction should be woven into the specification of both class `Point` and class `Line`.

6 Transforming Pipa Specifications Back to JML

One important reason to design Pipa based on JML is that we hope to make use of existing JML-based tools. To make this possible, we propose to develop a tool to automatically transform an AspectJ program together with its Pipa specification into a standard Java program and JML specification. To this end, we propose to modify the AspectJ compiler (`ajc`) to retain the comments associated with advice and aspect introduction during the weaving process. `ajc` supports Javadoc style and can retain comments of classes and interfaces during the weaving process. But this process does not retain comments of advice and introduction. By modifying the `ajc`, we can make it retain the comments of advice and introduction as well, with the resulting woven program a standard Java program with JML specifications. Therefore, after the transformation, all JML-based tools can be applied to AspectJ programs. However, when performing such transformations, we must find a way to handle certain problems such as those listed below.

The first problem is how to handle aspect invariants. A Pipa specification of a piece of advice or introduction can be directly transformed to a JML specification using a modified AspectJ weaver. However, the role of an aspect invariant in the weaving process is less clear, since aspect invariants may crosscut many different classes and should hold for all join points relevant to the advice. One conservative solution to this problem is to weave the aspect invariant into the class invariant of every class that the aspect crosscuts.

The second problem is how to handle the problem of specification weaving when weaving the aspects into classes. Several cases that are related to specification weaving must be considered because different advice may lead to different weaving rules. For example, a piece of after-returning advice, which is only valid for the case when the advised method returns normally, should not be woven into the specification of the advised method when the method returns by throwing an exception. Moreover, around advice that contains a `proceed()` construct also requires similar handling. However, for before advice or normal after advice, one can simply weave the specification into each method it advises with no additional adjustment. A transformation tool must correctly handle these different cases during the weaving process.

7 Related Work

Clifton and Leavens' [5] work on modular aspect-oriented reasoning also requires the specification of aspect advice. We see our work as differing from theirs in several ways. First, Pipa has a different goal. While the purpose of the Clifton and Leavens' work is to support modular aspect-oriented reasoning by extending

AspectJ with new language constructs, Pipa is intended to allow the full specification of AspectJ programs. Second, Pipa can specify the global properties of an aspect using aspect invariants that are different from pre- and postconditions for individual modules in the aspect. Third, Pipa supports aspect specification inheritance, and more importantly, Pipa supports aspect specification crosscutting, either structurally or behaviorally. Clifton and Leavens [5] focus on specifying advice in an aspect, not on issues about how to specify introduction, aspect invariant, aspect specification inheritance and crosscutting.

There has been significant work in the field of generic specification languages in general and BISLs in particular. Widely used generic specification languages include Z [21], VDM [11], and Larch [8]. Several BISLs that are based on Larch have been designed, each tailored to a specific programming language. Examples include LCL (for C) [8], LM3 (for Modula-3) [8], and Larch/C++ [4].

In addition to the Larch family, Meyer's work on the programming language Eiffel has advanced the cause of applying formal methods to object-oriented programs [18]. In Eiffel, unlike a Larch-style interface specification language, one can use Boolean expressions to specify pre- and postconditions for operations on abstract data types written in Eiffel, that is, program expressions can be used in pre- and postconditions. In addition, in Eiffel one can use class invariants to specify the global properties of instances of the class. On the other hand, several projects have been carried out to support the principle of Design By Contract (DBC), originally introduced by Meyer in Eiffel [18]. Examples include iContract [15] and Jass [3].

Recently, the emergence of Java as a popular object-oriented programming language has led to several BISLs designed for Java. Examples include JML [16], ESC/Java [6], and AAL [12]. JML allows assertions to be specified for Java classes and interfaces, and provide very expressive power to specify Java modules (classes and interfaces). ESC/Java is a static checking tool for Java. It can statically check for various errors in a Java program without executing the program. The annotation language in ESC/Java is a subset of JML that can be used to annotate Java code in various ways. AAL is an annotation language designed for annotating and checking Java programs. Like JML, AAL supports run-time assertion checking. AAL also supports full static checking for Java programs similar to ESC/Java. AAL translates annotated Java programs into Alloy [9], a simple first-order logic with relational operators, and uses Alloy's SAT solver-based automatic analysis technique to check Java programs. LOOP [10] is a project dedicated to verify JavaCard programs. LOOP adopts JML as its BISL for annotating Java modules and transforms annotated Java programs into a theorem-prover, PVS [20], to formally verify JavaCard programs.

Although the languages mentioned above can be used to specify programs written in various programming languages, they are not designed to specify programs written in AOP languages such as AspectJ. In summary, Pipa is the first BISL tailored to AspectJ that can be used to specify AspectJ programs. Pipa is also unique in its use of aspect invariants to specify the whole properties of an aspect, and in its supporting of aspect specification inheritance and crosscutting.

8 Concluding Remarks

In this paper we presented Pipa, a BISL tailored to AspectJ and discussed the goals of Pipa and the overall specification approach. Pipa is a simple and practical extension to JML. Pipa uses extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. We present several examples of Pipa specifications, and discuss how an AspectJ program together with its Pipa specification can be transformed into a corresponding Java program and JML specification, which is a crucial step towards the utilization of existing JML-based tools to verify AspectJ programs.

As future work, we would like to augment Pipa to support more kinds of join points, for example joint points at field accesses and dynamic join points such as `cflow` and `cflowbelow`. We also would like to refine our specification framework for AspectJ and to implement a tool to automatically transform an AspectJ program with Pipa specification into a corresponding Java program and JML specification.

Acknowledgements. We are grateful to Darko Marinov, Chandrasekhar Boyapati, and anonymous referees for valuable comments on an earlier version of the paper. We also thank Mik Kersten for valuable discussions on AspectJ implementation.

References

1. The AspectJ Team. The AspectJ Programming Guide. 2001. AspectJ home page: `http://www.aspectj.org`.
2. L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.
3. D. Bertetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - Java with Assertions. In K. Havelund and G. Rosu, editors, *ENTCS*, Vol. 55, Elsevier Publishing, 2001.
4. Y. Cheon and G. T. Leavens. A Quick Overview of Larch/C++. *Journal of Object-Oriented Programming*, Vol.7, No.6, pp.39-49, October 1994.
5. C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. Technical Report TR#02-04, Department of Computer Science, Iowa State University, March 2002.
6. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp.234-245, June 2002.
7. J. Gosling, B. Joy, and G. Steele. The Java Language Specification. The Java Series, Addison-Wesley, Reading, MA, 1996.
8. J. V. Guttag, J. J. Horning, S. J. Garland, K. D. Jones, A Modet, and J. M. Wing. Larch: Languages and Tools for Formal Specification. Springer-Verlag, New York, N. Y., 1993.
9. D. Jackson. Alloy: A Lightweight Object Modeling Notation. *ACM Transaction on Software Engineering and Methodology*, Vol.11, No.2, pp.256-290, April 2002.

10. B. Jacobs, J. van den Berg, M. Huisman, M. van Berkum, U. Hensel, and H. Tews. Reasoning About Java Classes (Preliminary Report). *Proc. ACM SIGPLAN 1998 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp.329-340, October 1998.
11. C. B. Jones. Systematic Software Development Using VDM. Prentice-Hall, Englewood Cliffs, N.J., second edition, 1990.
12. S. Khurshid, D. Marinov, and D. Jackson. An Analyzable Annotation Language. *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming Systems, Languages and Applications*, October 2002.
13. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *Proc. 11th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.
14. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and G. Griswold. An Overview of AspectJ. *Proc. 15th European Conference on Object-Oriented Programming*, pp.327-352, LNCS, Vol.2072, Springer-Verlag, June 2001.
15. R. Kramer. iContract- the Java Design by Contract Tool. *Proc. Technology of Object-Oriented Language and Systems (TOOLS-USA)*, 1998.
16. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary Design of JML: a Behavioral Interface Specification Language for Java. Technical Report TR98-06, Department of Computer Science, Iowa State University, 1998 (Last version: June 2002).
17. K. Lieberher, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.
18. B. Meyer. Object-Oriented Software Construction. Prentice Hall, New York, N.Y., Second Edition, 1997.
19. H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2001.
20. S. Owre, J. M. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, Vol.21, No.2, pp.107-125, February 1995.
21. J. M. Spivey. The Z Notation: A Reference Manual. Prentice-Hall, New York, N.J., Second edition, 1992.

Appendix: AspectJ

AspectJ [14] is a seamless aspect-oriented extension to Java by adding some new concepts and associated constructs to Java. These concepts and associated constructs are called join point, pointcut, advice, introduction, and aspect. We briefly introduce them in the following.

Aspect is modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect is defined by aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain state and methods, and also may be specialized in its sub-aspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. An aspect can *introduce* methods, attributes, and interface

implementation declarations into types by using the *introduction* construct. Introduced members may be made visible to all classes and aspects (public introduction) or only within the aspect (private introduction), allowing one to avoid name conflicts by using pre-existing members. In addition to introduction, the essential mechanism provided for composing an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. An aspect can specify *advice* that is used to define some code that should be executed when a pointcut is reached. Advice is a method-like mechanism which consists of code that is executed *before*, *after*, or *around* a pointcut. around advice executes *in place* of the indicated pointcut, allowing a method to be replaced. As a class, an aspect can also be declared as abstract, that means it can not be instantiated. By default, a concrete aspect has only one instance exists for the program execution. Also named pointcuts can be declared abstract within an abstract aspect, allowing them to be given concrete definitions within concrete sub-aspects, much as abstract methods are used.

An AspectJ program can be divided into two parts: *base code* part which includes classes, interfaces, and other language constructs, and *aspect code* part which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that the base and aspect code run together in a properly coordinated fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [1].

PacoSuite and JAsCo: A Visual Component Composition Environment with Advanced Aspect Separation Features

Wim Vanderperren, Davy Suvée, Bart Wydaeghe, and Viviane Jonckers

Vrije Universiteit Brussel, Pleinlaan 2
1050 Brussel, Belgium
{wvdperre,dsuvee,bwydaegh,vejoncke}@vub.ac.be
http://ssel.vub.ac.be

Abstract. This paper presents the visual component composition environment called PacoSuite and the tools needed for the JAsCo aspect-oriented programming language. PacoSuite allows plug-and-play component composition without in-depth technical knowledge of the components. PacoSuite uses three constructs: components, composition patterns and composition adapters. A composition pattern is an abstract and reusable description of a collaboration between components. A composition adapter on the other hand, describes transformations of a composition of components and is used to modularize crosscutting concerns. A composition adapter is able to have an implementation in the JAsCo language in order to invasively alter components. Compatibility of a given collaboration is checked using finite automaton theory and the glue-code to make the composition work is generated automatically.

1 Introduction

Current practice visual component composition environments are still far from reaching the plug and play ideal promised by component based development [4]. Expert technical knowledge of components is required in order to be able to compose them. There's no support whatsoever to verify whether a given composition of components is able to work together. Glue-code still has to be written manually to achieve a more involved collaboration than a mere event-method connection. Moreover, most tools don't even support reusing such a simple collaboration. To solve these problems, we propose a novel visual component composition environment, called PacoSuite. PacoSuite lifts current practice component composition to a higher abstraction level. Composition patterns are introduced as reusable and abstract collaborations. Composition patterns as well as components are documented by a special kind of MSC [2]. PacoSuite automatically validates a given composition using finite automaton theory. In addition, glue-code which enables the collaboration is generated. Recently, composition adapters have been introduced to separate crosscutting concerns [3] that do not fit into our current constructs. Composition adapters describe transformations of a composition pattern independent of a specific API. In addition, a composition adapter is able to have an implementation in the JAsCo

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 166–169, 2003.

aspect-oriented implementation language. This enables a composition adapter to influence the interior behavior of components. We refer to [5,6,7] for more information on the fundamentals of this approach.

The next section describes the PacoSuite tool in more detail. Section 3 shortly sketches the tools required by the JAsCo aspect-oriented programming language.

2 PacoSuite

PacoSuite consists of two tools: a visual documentation editor called PacoDoc and the actual component composition environment called PacoWire. Both tools are written in the Java language. PacoDoc allows the user to construct usage scenarios, composition patterns and composition adapters in a user-friendly manner. Afterwards, the drawn diagrams are exported to an XML file. PacoDoc is also integrated in PacoWire, such that a component composer is able to view the documentation of a component at any time. Fig. 1 shows a screenshot of PacoDoc.

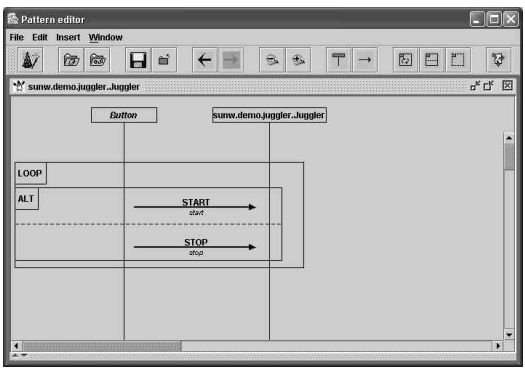


Fig. 1. The documentation of the Juggler component shown in PacoDoc

PacoWire is our actual component composition tool and contains a set of components, composition patterns and composition adapters nicely sorted into different categories. Creating an application is as simple as visually dragging components onto a composition pattern. The drag is refused when a component is detected to be incompatible with the selected composition pattern. Fig. 2 illustrates a screenshot of PacoWire where the component composer is about to drag the juggler component onto the subject role of the ToggleControl composition pattern. The ToggleControl composition pattern specifies a toggling behavior (consecutive starts and stops). The subject role receives the commands and the control role is responsible for sending the toggle commands. As the Juggler component is able to receive consecutive start and stop commands (see Fig. 1), it can fulfill the subject role of the ToggleControl composition pattern. However, when the component composer would drag the Juggler component onto the control role, the drag would be refused because the Juggler component can only receive messages. The JButton component from the Java Swing library for

instance, is compatible with the control role. After the JButton is dragged onto the control role, glue-code that implements this collaboration can be generated. The resulting application allows the juggler to be toggled from a single button. Notice that it is impossible to visually wire even this simple collaboration in current component composition environments because state information is required.

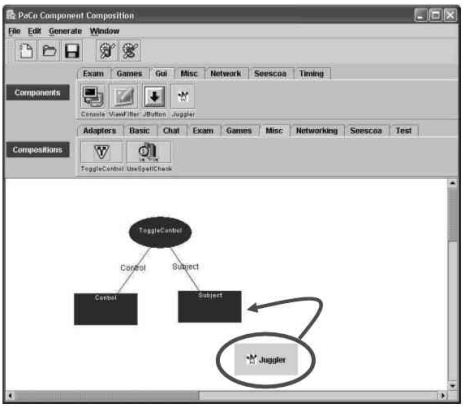


Fig. 2. Screenshot of PacoWire. The component composer is about to drag the Juggler component onto the subject role of the ToggleControl composition pattern

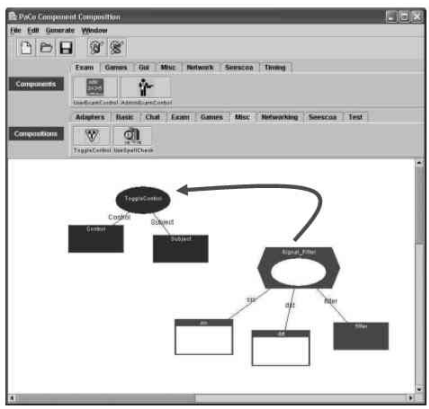


Fig. 3. Applying the invasiveTimer composition adapter onto the ToggleControl composition pattern

Applying a composition adapter is also achieved by a simple drag and drop. The tool takes care of inserting the transformations the composition adapter describes. When the composition adapter is implemented using JAsCo, the JAsCo tools are executed transparently for the user. In fact, a component composer doesn't even have to know whether a composition adapter has a JAsCo implementation or not. Fig. 3 illustrates a screenshot of PacoWire, where a component composer is about to map the SignalFilter composition adapter onto the ToggleControl composition pattern. The SignalFilter composition adapter describes a logging aspect. The communication between the source and destination roles is trapped and re-routed through the filter role. In this way, the filter role is able to log the events the component composer is interested in. To apply the SignalFilter composition adapter onto the ToggleControl composition pattern, the component composer can simply drag one onto the other. The source and destination roles of the SignalFilter composition adapter are then automatically mapped onto roles of the composition pattern using an algorithm based on dynamic programming ideas [1]. The tool issues a warning when the application of this composition adapter onto the selected composition pattern is not valid. Afterwards, the JButton and Juggler components are mapped onto the roles of the ToggleControl composition pattern as before. A logging component that writes received events onto disk can for instance be mapped onto the filter role. When the glue-code is generated, the juggling application works just as it did before. However, every signal from the button is first rerouted through the logging component before it is sent to the Juggler component.

3 JAsCo

The JAsCo-language has been implemented to allow composition adapters to affect the internal behavior of components. The JAsCo-framework provides 4 tools which are required to deploy aspects on components.

The key tool of the JAsCo-package is the *BeanTransformer*. To enable interaction between aspects and components, we propose a new component model where each public method of a component is provided with a *trap*. These traps reroute control-flow at run-time, which enables the execution of aspect behavior. The *BeanTransformer*-tool is responsible for transforming a regular Java Bean into a JAsCo bean component. This tool employs state-of-the-art Java byte code adaptation techniques for inserting *traps* at the appropriate places.

The JAsCo-language itself stays as close as possible to the regular Java syntax and introduces two concepts: aspect beans and connectors. Aspect beans are used for describing crosscutting behavior. Deploying an aspect bean within an application is done by making use of connectors. The definition of both aspect beans and connectors is preprocessed to a Java source code file. Afterwards, this definition is compiled to its Java class-representation by making use of the standard Java Compiler. Both the *CompileAspect*- and *CompileConnector*-tool are responsible for managing this compilation-process.

The fourth tool contained within the JAsCo-package is the *Introspector*-tool, which is a GUI environment that allows introspecting what connectors are loaded. Connectors can be added and removed at run-time, which enables to dynamically change the properties of the system. The tool displays the various hooks that are instantiated by the connectors and the targets on which these hooks are applied.

References

[1] Bellman R.E. & Dreyfus S.E. Applied Dynamical Programming. Princeton University Press, 1962.

[2] ITU-TS. ITU-TS Recommendation Z.120: Message Sequence Chart (MSC). ITU-TS, Geneva, September 1993.

[3] Kiczales, G., Lamping, J., Lopes, C.V., Maeda, C., Mendhekar, A. and Murphy, A. Aspect-Oriented Programming. In proceedings of the 19th International Conference on Software Engineering (ICSE), Boston, USA. ACM Press. May 1997.

[4] Short, K. (1997). Component Based Development and Object Modeling. Available at: `http://www.cool.sterling.com/cdb/whitepaper/2.htm`

[5] Suvée, D., Vanderperren, W., and Jonckers, V. JAsCo: an Aspect-Oriented approach tailored for CBSD. In Proc. of AOSD int. Conf., Boston, USA, march 2003.

[6] Vanderperren, W. Localizing crosscutting concerns in visual component based development. In proc. of SERP international conference, Las Vegas, USA, june 2002.

[7] Wydaeghe, B. and Vandeperren, W. Visual Component Composition Using Composition Patterns. In Proceedings of Tools 2001, July 2001.

Model-Based Development of Web Applications Using Graphical Reaction Rules

Reiko Heckel and Marc Lohmann

Faculty of Computer Science, Electrical Engineering and Mathematics
University of Paderborn
D–33098 Paderborn, Germany
{reiko,mlohmann}@upb.de

Abstract. The OMG's Model-Driven Architecture focusses on the evolution and integration of applications across heterogeneous middleware platforms. Presently available instances of this idea are mostly limited to static models.
We propose a model-driven approach to the development of web-enabled applications, seen as reactive information systems on an HTTP-based communication platform, which covers both static and dynamic aspects. To support the separate change of both platform and functionality we separate at model and implementation level the platform-independent application logic from classes specific to technologies like HTML or SOAP.
We discuss a notion of consistency between models at different abstraction levels based on a concept of graphical reaction rules, i.e., graph transformation rules which integrate data state transformation and reactive behavior.

1 Introduction

Most business applications developed today depend on a specific middleware platform providing services for communication, persistence, security, etc. while supporting interoperability across different kinds of hardware and operating systems. If such systems have to interact over the web, for example to provide integrated services, we face the problem of the interoperability of these platforms. Solutions at different levels have been proposed to overcome this problem.

At implementation level, the approach of web services provides a collection of languages and protocols to support interoperability at the level of text-based HTTP by interchanging XML-documents representing, e.g., remote procedure calls. At the level of design, the OMG has proposed the Model-Driven Architecture (MDA) [16,17] to achieve interoperability through models. Starting from standard UML models [12] specifying the intended functionality, the MDA approach is largely concerned with the vertical structure of the mappings required to implement this functionality on any given platform. The idea is to distinguish between *platform-independent models* (PIMs) that are refined into *platform-specific models* (PSMs) which carry all relevant annotations for the generation of platform-specific code.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 170–183, 2003.

The idea seems realistic (and has in part been implemented) for mappings to data type or interface definition languages like SQL DDL, XML Schemata, or CORBA IDL, and for static aspects of Java and EJB, thanks to the relative simplicity and stability of these more mature languages. Siegel [15], for example, describes the use MDA for applications based on web services, focussing on the integration of service interfaces in applications developed with the MDA approach. However, the integration does not take into account dynamic information about how the services should be used.

For behavioral models like statecharts and collaboration diagrams, transformations to code are more sophisticated, but largely platform-independent [3,5]. If we want to use such diagrams to model the business logic of our application, an integration with the platform-specific models and their mappings is required.

According to the OMG's proposal this integration should be done by augmenting, throughout the platform-independent models, model elements with annotations in terms of stereotypes and tagged values. Then, in the likely case of a change in the platform-independent model (induced, for example, by an evolution of the functional requirements), the platform-specific annotations have to be re-iterated for all relevant target platforms. This shows an imbalance of the MDA proposal which focuses entirely on evolution and integration across platforms, disregarding the evolution of the functional aspect.

In this paper, we discuss a model-driven approach to the development of web-enabled applications which avoids the above obstacles by separating platform-independent and platform-specific models. This separation of concerns is preserved at the implementation level as well as in the mapping.

The approach aims at both interactive (HTML-based) web applications and web services which share the basic request-query/update-response pattern, where an HTTP-request is answered (or causes further requests to other servers) in combination with a query or update of the internal data state of the server or its associated data base. Rather than with other kinds of reactive systems, like in the embedded domain where data is often abstracted from at the level of models, web-based business applications are primarily information systems, and data state transformations are a crucial aspect of their behavior.

Our approach to integrate data state transformations and reactivity at the level of models is based on a special format of graph transformation rules that we call *graphical reaction rules*. Such a reaction rule is a transformation rule on the internal object structure of the server which is triggered by a *request*, i.e., a special kind of vertex modeling an incoming message that is consumed when the rule is applied. This rule-based way of formalizing reactive behavior goes back to actor grammars [10], a model of actor systems by graph transformation, and has since then been adopted by several authors.

Based on this concept we outline a model-based development approach starting from requirements expressed in terms of use cases and sequence diagrams in Sect. 2 via architectural and detailed platform-independent design in Sect. 3 to platform-specific design and implementation in Sect. 4. Then, in Sect. 5, we provide a discussion of the vertical consistency problems arising between these

different levels and argue for the necessity of a model-driven testing approach in Sect. 6.

2 Requirements Specification

Following the Unified Process [9], the functional requirements of web-enabled applications are captured by use cases where interesting scenarios are detailed by means of sequence diagrams.

As a running example throughout the paper we use the model of an online shop. As shown by the use case diagram in Fig. 1, a client of the online shop can query products, order a product, or pay for an order. If the client wants to pay, for example by credit card, his credit card data has to be verified before he gets an acknowledgement. Therefore, the online shop uses the service of a credit card company to verify credit card data.

Sequence diagrams are used to model scenarios like this in a more formal way as sequences of messages exchanged, in this case, between the client, the online shop, and the credit card company. Variants can be expressed by different sequence diagrams associated with the same use case. Figure 2 shows two sequence diagrams detailing the use case *pay order*. The initial segments of both scenarios are identical: The client who triggers the use case is asked by the online shop to enter his preferred method of payment, e.g., by automatic debit from the client's bank account or by credit card. In our sample scenarios the client chooses to pay by credit card, which requires the transmission of the credit card data, e.g., the name of the credit card company, the credit card number, etc. The online shop sends the data to a credit card company for validation. The client gets a positive or negative feedback, depending on whether the credit card check has been successful or not. That means, we have one success and one failure scenario.

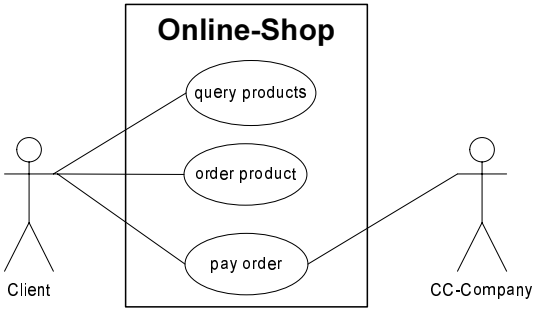


Fig. 1. Use case diagram of the shop example

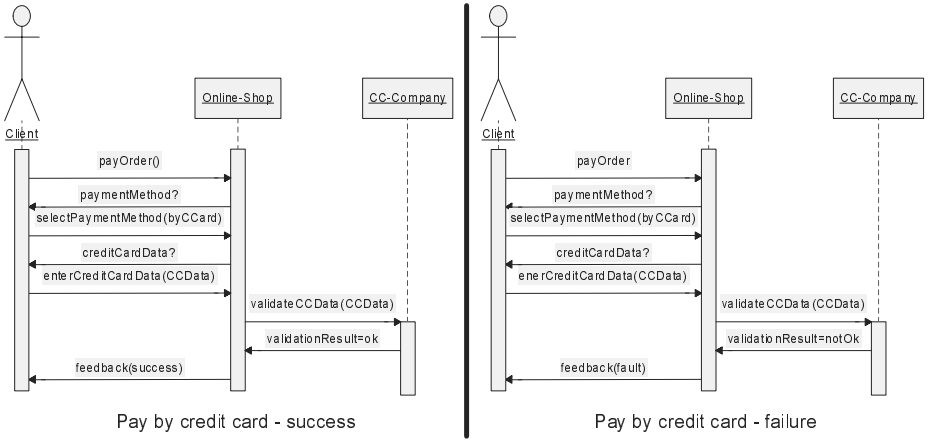


Fig. 2. Two scenarios describing the use case pay order

3 Platform-Independent Design

The requirements specification presented in Sect. 2 focuses on specific scenarios of the externally visible behavior of the application. In this section, we will complement this view, first moving from specific examples to general specifications and then from external requirements to the internal realization. The first step, called *architectural design*, describes the involved components and their possible interactions. This model is refined in the *detailed design* adding internal data state transformations.

3.1 Architectural Design

The structural view of the architectural design describes the components and interfaces of the online shop. We provide an interface for every use case, which is later implemented by a control class to execute the use case. Figure 3 shows the three components of our online shop. The online shop itself is a component with three interfaces for the three use cases in the diagram of Fig. 1. For each interface we can list the operations of the online shop that can be called by a client through that interface. In Fig. 3 we have only detailed the interfaces for the use case *pay order* and for the *card validation* use case of the credit card company.

The data objects exchanged as parameters of operation calls between client and online shop as well as between online shop and credit card company are specified in a class diagram. We are qualifying the corresponding classes with stereotypes, one of the extensibility mechanisms of UML. The general notation for the use of stereotype is to enclose it in guillemots (≪≫). We are using the stereotype *boundary* defined within the Unified Process [9] which is intended to designate GUI classes or forms that model the interaction between the system and its actors.

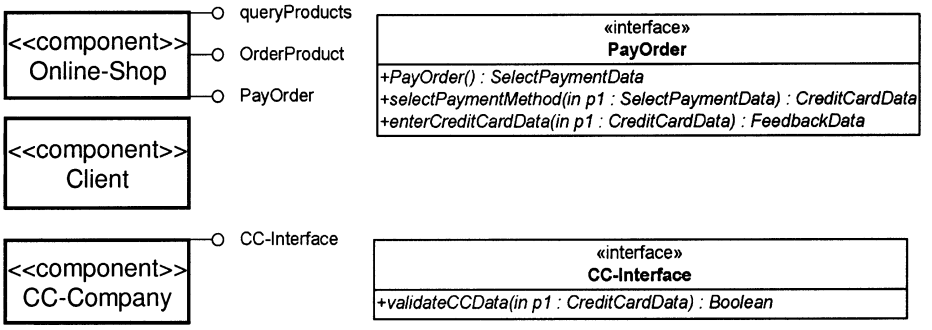


Fig. 3. Components of the online shop example

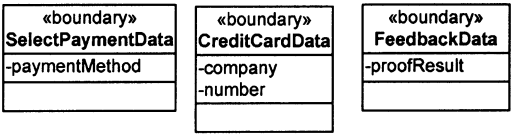


Fig. 4. Boundary classes describing the exchanged data

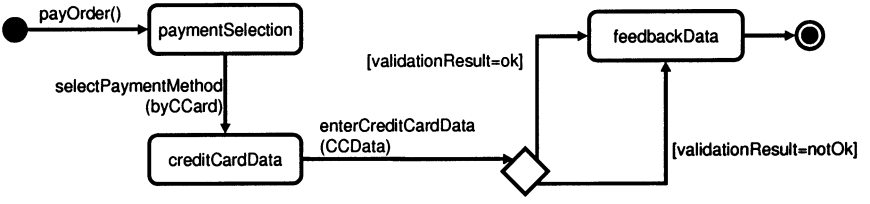


Fig. 5. Protocol statechart for the component online shop

Whereas sequence diagrams are used to describe single scenarios from a global point of view, protocol statecharts are used to specify the sequences of requests individual components are willing to accept. Figure 5 shows the statechart for the interface *payOrder* corresponding to the use case pay order. The states are named according to the boundary classes whose instances are transferred to the client in response to the previous request.

3.2 Detailed Design

After having described components from an outside perspective, in this section their data structures and computations are modelled.

Class diagrams are used to represent static aspects. Figure 6 shows the result of detailing the use case *pay order*. Beside the stereotype *boundary* introduced above, *control* and *entity* stereotypes are used (cf. again [9]): Each of these

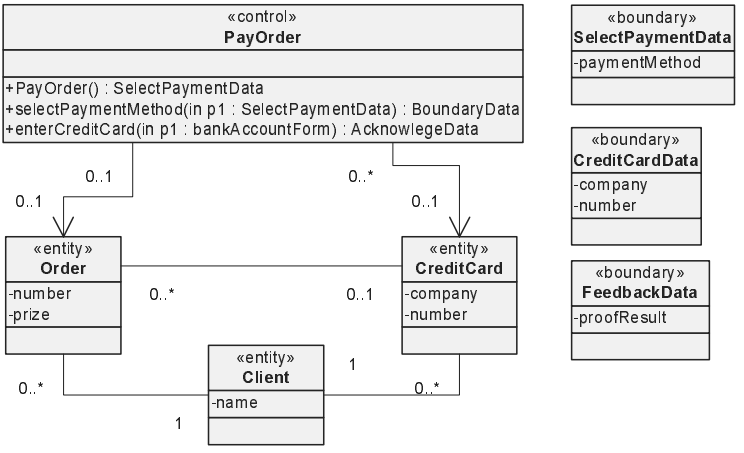


Fig. 6. Platform-independent class diagram for online shop

stereotypes expresses a different role of a class in the implementation. Instances of control classes coordinate other objects, encapsulating the control related to a specific use case. Boundary classes are used to model interactions between the system and its actors. Entity classes model long-lived or persistent information.

In addition to the static and dynamic diagrams of our models we now introduce a *functional view* integrating the other two by describing the effect of an operation on the data. This requires to move the focus both from sequences to single operations and from the externally visible behavior to its internal realization. To simplify this transition, we first take an operation-wise view on the externally visible behavior. To this aim we introduce for each operation *abstract reaction rules*, which are derived from the sequence diagrams. For example, Fig. 7 shows the abstract reaction rules for the success scenario of Fig. 2. The left-hand side of the rule contains the method call on the component as part of the precondition of the operation. The right-hand side shows the visible effects, i.e., a message sent to another component.

Next, the internal data state transformation associated with this operation is described. Figure 8 shows the refinement of the lower right abstract reaction rule of Fig. 7. The left-hand side of the diagram represents the precondition of the rule, i.e., that the validation of the credit card has been successful. The right-hand side shows the desired effect of the execution: A new boundary object with an acknowledgement is created.

One benefit of this form of specification integrating static and dynamic models is that it provides a detailed and precise enough specification to allow automatic code generation (cf. [3]) which is an essential for the goal of model-driven development.

To stick with standard UML notation, reaction rules can be expressed as collaboration diagrams where pre- and post-conditions are jointly represented in one diagram. In order to distinguish objects that are created or deleted, the stan-

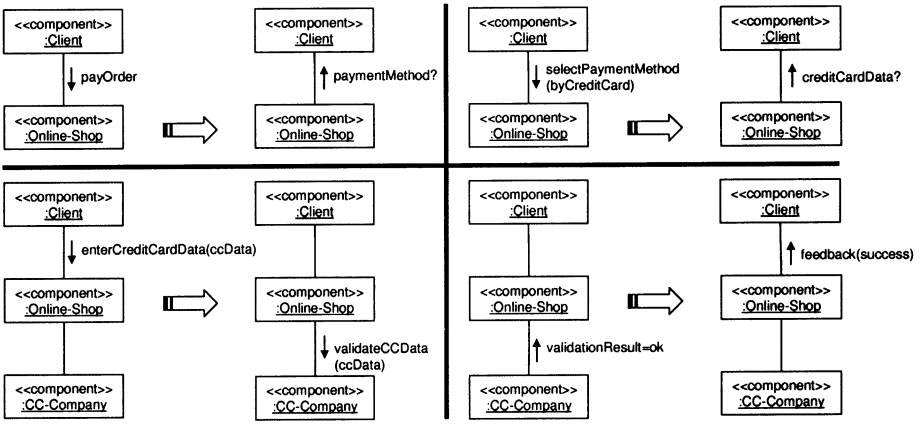


Fig. 7. Abstract reaction rules

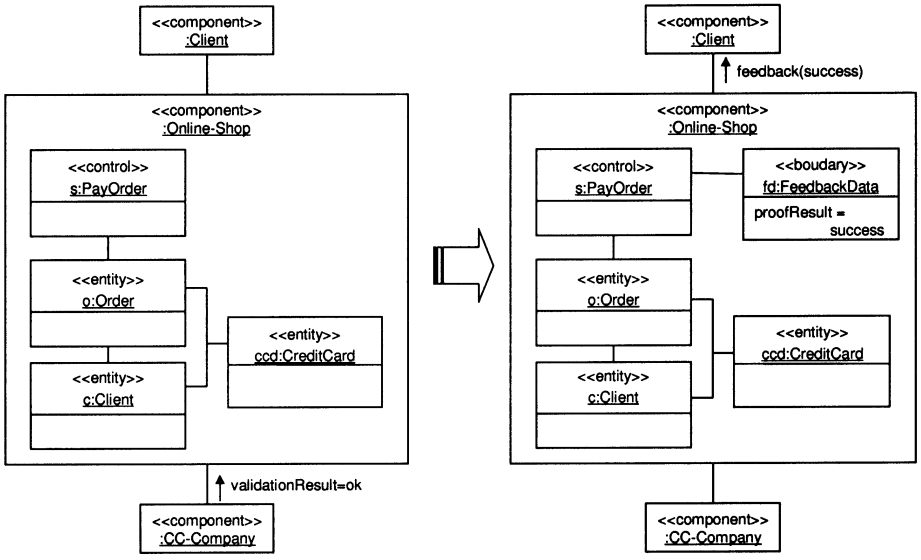


Fig. 8. Refinement of lower right abstract reaction rule of Fig. 7

dard constraints *new* and *destroyed* are used. As an example, Fig. 9 shows the corresponding representation of the reaction rule of Fig. 7. This correspondence between collaboration diagrams and graph transformation rules can be extended to more sophisticated cases where complex scenarios are modelled within one diagram [6]. In this case, rather than a single rule, a *graph process* is required, i.e., a partially ordered set of interrelated rules each modeling a single step of the interaction.

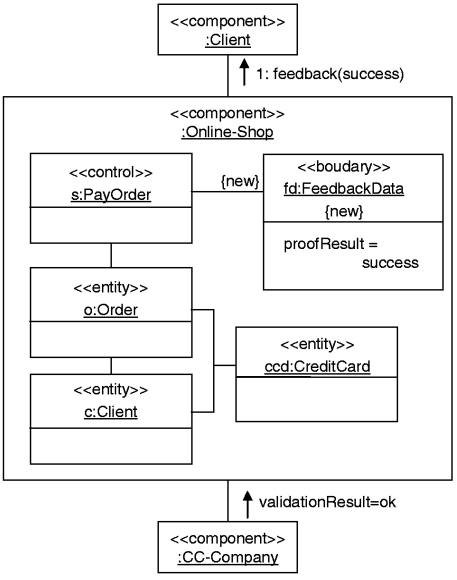


Fig. 9. Reaction rule of Fig. 8 expressed with collaboration diagram

4 Integrating Platform-Independent and Platform-Specific Models

In the next two subsections we show how to map the platform-independent models developed in the previous section on platforms like HTML or SOAP which realize the request-query/update-response pattern. Model information required for this mapping is captured in the *platform-specific design*. The aim is to deploy code generated from platform-independent models on a specific middleware while keeping platform-independent and platform-specific models separated.

4.1 Platform-Specific Design

In many web-enabled applications a middleware serves as a link between clients and back-end services. This middleware is normally responsible for the implementation of the chosen base technology. For example you can use a web server which implements the Java Servlet technology [18] to implement an HTML application or you can use a SOAP server to implement a SOAP-based [19] application. In both cases a client doesn't send a request directly to the application. Instead a client addresses the middleware and this middleware requests the application with pre-defined interfaces.

Figure 10 shows an abstract overview how a web application realized with the Java Servlet technology works. A user normally fills in an HTML form on the client and by clicking the Submit button the form data is send to the server (1). The server locates the requested application, more exactly he locates an

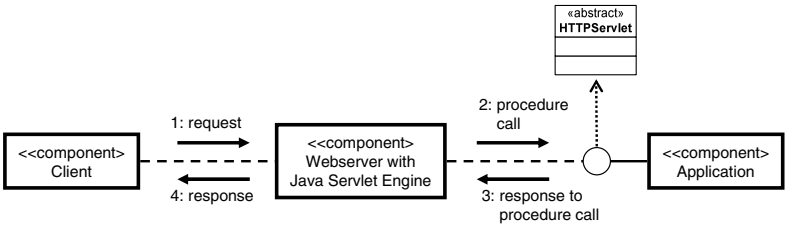


Fig. 10. Using a servlet (abstract overview)

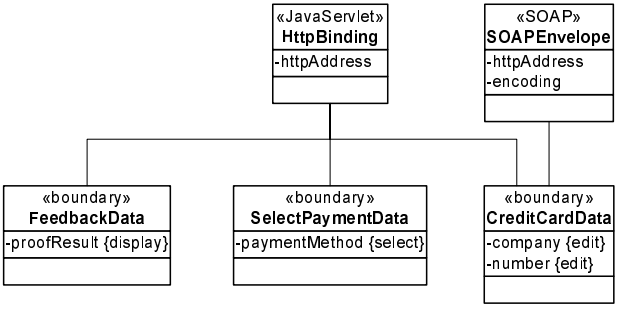


Fig. 11. Annotated boundary classes for interactive HTML application and for a SOAP Service

application which implements the servlet interface. The application is called via the servlet interface by a normal method call (2). The application processes the data and calculates a response (3). The middleware (servlet engine) transfers this response back to the client (4). That means to implement a Servlet, a servlet API is used, especially the Servlet interface. The Servlet interface declares, but does not implement, methods that manage the servlet and its communications with clients. These interfaces have to be provided when developing a servlet [18]. Other Internet-based communication platforms, like the SOAP implementation Apache Axis [13], work in a similar way.

To use different kinds of middleware, we have to annotate the boundary classes with information needed by the middleware. Figure 11 shows how this annotation could look like. For using the Java servlet technology we annotated the boundary classes with an additional class *HTTPBinding*. This class contains an attribute for the address where the application is to be found. Further, we have to annotate the attributes of the boundary classes to show, which information is displayed to the client and which information can be edited by the client. Therefore, we use property strings, marked with curly brackets. Property strings indicate property values that apply to an attribute and an attribute can contain more than one property value. *Edit* means that the client can edit the data freely, *select* that the client can choose one value from many and *display* that data is only transferred to the client. To access the credit card institute via SOAP the

class *CreditCardData* also needs some SOAP specific annotations. Objects are encoded in a special way in a SOAP message and the message has to be sent to an appointed web address.

4.2 Implementation

In this section we describe the integration of the application logic generated from the platform-independent model with the platform-specific technology.

Figure 12 shows the integration for the Java Servlet technology. Classes that are specific to the technology are shown in grey. Let us at first explain the changes of the platform-independent models made during the code generation. In Figure 12 a new platform-independent abstract class *StatemachineHandler* is shown which has the task of an object controller. For every use case, an implementation for the abstract method of this class is generated. The method implements the statechart of the corresponding use case, filtering incoming requests according to the protocol. As a result, the methods of the class *PayOrder* are never called directly. Instead, every time a method of the class *PayOrder* has to be called, the method *executeMethod* is called, which calls the correct method depending on the state and the type of the incoming boundary data. This design is following the *Command* design pattern [4], which allows a decoupling between the sender and the receiver. Decoupling means that the sender has no knowledge of the receiver's interface.

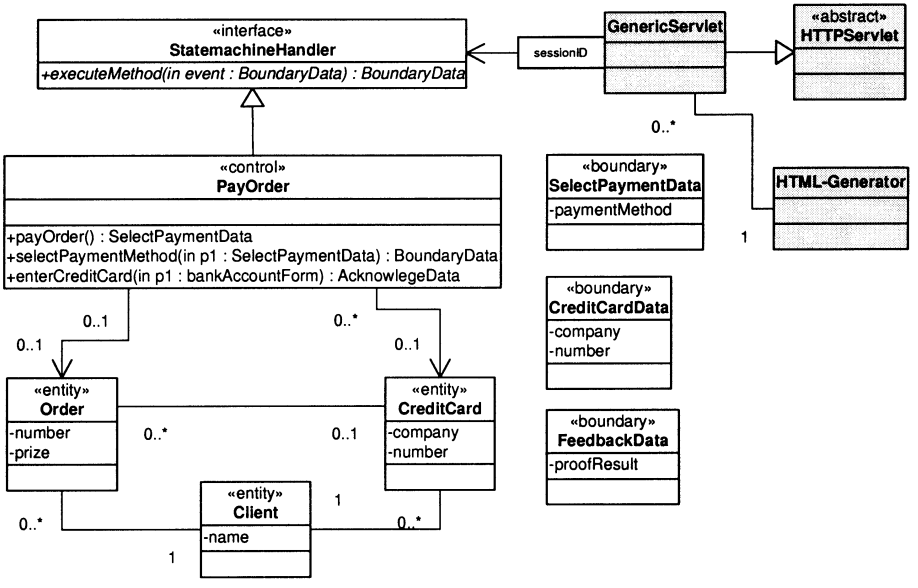


Fig. 12. Integration of PIM and PSM on implementation level

On the platform-dependent side the elementary class *GenericServlet*, which has to be implemented by a software developer, inherits from an abstract class *HTTPServlet*, to allow integration in a Java Servlet-based server. This platform-dependent class calls the platform independent classes realizing the application logic. The class *GenericServlet* is called when a client has filled in a form, for example an HTML representation of the platform-independent boundary class *selectPaymentForm*. It then calls the corresponding method of a control class of the platform-independent model. Therefore, some auxiliary information has to be encoded in the forms transmitted to the client, because no control information of the application shall be encoded in the platform-dependent classes. In our example, we need a *sessionID* to obtain the correct session object for each request from a client.

In Fig. 12 we have one more platform-specific class which has to be implemented by a software engineer. The class *HTML-generator* is responsible for the creation of an HTML form from the boundary classes, which is sent to the client. To create an HTML form from a specific boundary class the platform specific annotations (see Fig. 11) are needed.

To use alternative platforms or other kinds of user interfaces, only the platform specific classes of Fig. 12 have to change. For example, to use XForms [20], a new technology more powerful than HTML forms and based on XML, only the *HTML-Generator* has to be changed to create this new kind of user interface. Further you have to ensure, that the middleware calling the servlet is able to evaluate XForms. For other middleware, one may have to replace the class *GenericServlet* by another class which implements the required interface of the technology.

5 Vertical Consistency

The quality of models is a prerequisite for the quality of the resulting system. One important aspect of quality (and one of the few that can be sufficiently formalized) is consistency. In general, consistency problems occur if different views of the same system are redundant or dependent on each other. Depending on whether the views are at the same or at different levels of abstraction, we distinguish horizontal and vertical consistency problems, respectively.

Vertical consistency is a property of the transition from requirements to design models and their implementation. As such, it is a property responsible for the transition between platform independent and platform-specific models in the MDA approach. Therefore, in this paper we concentrate on this aspect, and in particular on the consistency of behavioral models. In this category we face two vertical consistency problems.

1. Requirements expressed in terms of scenarios in Sect. 2 have to be consistent with the contracts between service provider and client as specified by the protocol statecharts of the architectural design in Sect. 3.1.

2. These contracts (protocols) have to be fulfilled (correctly implemented) by the components providing the services, as described by the reaction rules of the detailed design in Sect. 3.2.

The first consistency requirement relates the sequence diagrams in Fig. 2 and the protocol statechart in Fig. 5: The sequence of requests (incoming messages) received by the Online Shop component instance in any of the two diagrams must be acceptable by the statechart diagram (or they must be subsequences of an acceptable sequence if we want to allow for sequence diagrams showing only a part of the possible behavior). This is the case in both examples, but if we remove, e.g., the *selectPaymentMethod(byCCard)* message, the consistency requirement is violated.

To validate this notion of consistency, it may be phrased as a model checking problem by translating statecharts and sequence diagrams into a common semantic domain, like CSP [7], which allows to express and check the condition that every trace of requests to a service contained in a sequence diagram is (an instance of) a subsequence of a trace of call events generated by the protocol statechart of that service (cf. [1]).

The second problem is concerned with the correct implementation of the protocols by the internal data state transformations of reaction rules. Given an initial data state for a component, we may ask if all sequences of requests are indeed executable in the sense that the current internal data state satisfies the precondition of all reaction rules that may be applied at this step according to the protocol. For example, the rule in Fig. 8 requires in its precondition the presence of an object of class *CreditCardData*. If this is not available, this rule is not applicable and the execution of the whole sequence fails.

To make this notion of consistency precise, we need to exercise our reaction rules in all possible ways prescribed by the protocol statechart. This is possible because of the formal background of graph transformation which provides us with an operational semantics for such rules, i.e., a notion of *graph transformation* which, abstractly speaking, defines a binary relation on states induced by rule applications. (See, e.g., [14] for different ways of formalizing this concept.)

To implement the protocol defined by the statechart of the component, the transformation relation thus defined must be able to produce a superset of the traces obtained from the statechart. This linear-time condition could be replaced by more sophisticated notions, e.g., using the failure and divergence model [7] or various notions of simulation on transition systems. A detailed study of what is a semantically convincing and at the same time feasible approach is outside the scope of this paper. One advantage of the simpler, linear condition is that it can be validated by testing.

6 Towards Model-Driven Testing

In order to test the consistency between models, an execution of models is required either by a model interpreter or through a model compiler which translates models into executable code. Examples of the former include statechart

simulators like [8], but also graph transformation engines like AGG [2]. Compilers of statecharts can be found, for example, in the Rhapsody [5] and Fujaba [3] CASE tools, while the latter also transforms graph transformation rules (denoted as UML collaboration diagrams) into executable Java code.

In the context of MDA, a model shall be mapped on multiple platforms, thus reusing the effort of coding and design, but not the amount of testing required because implementations obtained from the same model may behave differently on different platforms. Hence, we require what could be called an approach to model-driven testing. By this we mean the testing of consistency properties, among models or between models and code, while reusing the results of platform-independent tests (or of tests performed on an "ideal" platform, like a single Java virtual machine) for implementations of the same models on different or heterogeneous target platforms. The idea is that recording platform-independent test results, we determine the expected results of platform-specific tests, which can then be automatically executed and compared. This idea is independent of the question whether the original tests are performed automatically or by hand.

7 Conclusion

In this paper we have presented a model-driven approach to the development of reactive information systems based on web technology which refines the OMG's MDA in order to separate better the technology-independent from the platform-specific aspects. We have used graphical reaction rules to specify reactive behavior in combination with data state transformation and discussed the consistency issues arising with more abstract protocol specifications by means of statecharts and requirements expressed by sequence diagrams.

Although graphical reaction rules are not part of the mainstream UML methodology, our experience with the use of this concept in a course on web-based application development at the University of Paderborn suggests that the higher level of integration of static and dynamic aspects adds to the understandability of models.

Future work shall include the development of tool support for automated consistency checks based on the construction of labeled transition in the previous section, as well as the exploration of the idea of model-driven testing.

References

1. G. Engels, J.M. Küster, L. Groenewegen, and R. Heckel. A methodology for specifying and analyzing consistency of object-oriented behavioral models. In V. Gruhn, editor, *Proc. European Software Engineering Conference (ESEC/FSE 01), Vienna, Austria*, volume 1301 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`, pages 327–343. Springer Verlag, 2001.
2. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 551–601. World Scientific, 1999.

3. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`. Springer-Verlag, 2000.
4. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addsion-Wesley, 1994.
5. D. Harel and E. Gery. Executable object modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.
6. R. Heckel and St. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, volume 2185 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`. Springer-Verlag, April 2001.
7. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland*. Carleton Scientific, 2000.
9. I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
10. D. Janssens and G. Rozenberg. Actor grammars. *Mathematical Systems Theory*, 22:75–107, 1989.
11. R. Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proc. 12th Intl. Conference on Concurrency Theory (CONCUR 2002), Aalborg, Denmark*, volume 2154 of *Lecture Notes in Comp. Science* `http://www.springer.de/comp/lncs`, pages 16–35. Springer-Verlag, August 2001.
12. Object Management Group. UML specification version 1.4, 2001. `http://www.celigent.com/omg/umlrtf/`.
13. The Apache XML Project. Axis user's guide. `http://xml.apache.org/axis/`, 2002.
14. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
15. Jon Siegel. Using omg's model driven architecture (MDA) to integrate web services, 2002. `http://www.omg.org/mda/mdafiles/MDA-WS-integrate-WP.pdf`.
16. Jon Siegel and OMG Staff Strategy Group. Model driven architecture, revision 2.6, November 2001. `ftp://ftp.omg.org/pub/docs/omg/01-12-01.pdf`.
17. Richard Soley and OMG Staff Strategy Group. Model driven architecture, draft 3.2, November 2000. `ftp://ftp.omg.org/pub/docs/omg/00-11-05.pdf`.
18. Sun Microsystems Inc. Java(tm) servlet specification 2.3. `http:://java.sun.com/products/servlet`, 2001.
19. W3C. Soap version 1.2 part 1: Messaging framework. `http://www.w3.org/TR/2002/WD-soap12-part1-20020626/`, 2002.
20. W3C. Xforms 1.0 W3C candidate recommendation. `http://www.w3.org/TR/2002/CR-xforms-20021112/`, November 2002.

Modular Analysis of Dataflow Process Networks

Yan Jin[1], Robert Esser[1], Charles Lakos[1], and Jörn W. Janneck[2]

[1] School of Computer Science, University of Adelaide, SA 5005, Australia
Fax: +61 8 8303 4366
{yan,esser,charles}@cs.adelaide.edu.au
[2] EECS Department, University of California at Berkeley, CA 94720-1770, USA
Fax: +1 510 642 2739
jwj@acm.org

Abstract. Process networks are popular for modelling distributed computing and signal processing applications, and multi-processor architectures. At the architecture description level, they have the flexibility to model actual processes using various formalisms. This is especially important where the systems are composed of parts with different characteristics, *e.g.* control-based or dataflow-oriented. However, this heterogeneity of processes presents a challenge for the analysis of process networks. This research proposes a lightweight method for analysing properties of such networks, such as freedom from unexpected reception and deadlock. The method employs interface automata as a bridge between the architectural model and heterogeneous processes. Thus, the properties are determined by a series of small tasks at both the architecture level and the process level. This separation of concerns simplifies the handling of heterogeneous processes and alleviates the potential state space explosion problem when analysing large systems.

1 Introduction

In recent years, component-based development has emerged as a significant factor in the production of large-scale software applications. By building systems from independently developed components, a promising means of achieving software reuse, rapid development and complexity management is provided.

Typically, components are black-box entities that encapsulate services behind their interfaces. The specifications of these interfaces tend to be rather limited, often capturing only the *signatures* of components, i.e. the names, data types and direction of ports excluding any information about communication *protocols*. Even with additional informal descriptions, such specifications are not adequate for designing reliable and evolving software systems. Instead, more rigorous specifications are needed, which capture interface behaviours of components, including the services that a component provides, the information about how it can be properly deployed, and the dependencies between its inputs and outputs. Naturally these specifications must not disclose implementation details of their components.

Having suitable specifications for the components is only part of the story — it is also necessary to provide flexible composition schemes. Direct composition is often difficult and sometimes impossible [1]. Instead, it is preferable to provide flexible connectors so

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 184–199, 2003.

that component-based systems can be constructed using various design strategies [2], where suitable architectural styles, *e.g.* pipe-and-filter and client/server architectures, can be employed. The major challenge is then to ensure that the resulting systems are consistent (namely, all components are properly deployed in the design) and that these systems meet global functional and nonfunctional requirements such as structural invariants, reliability and security.

This paper presents a step towards the component-based development and modular analysis of dataflow process networks. Such networks support flexible interconnection strategies as mentioned above. In our presentation, components (or processes) communicate through their input and output ports and the interconnection of components specifies a causality relation between data flow through input/output ports of the components.

In order to avoid the state space explosion problem, the composition of components is not analysed directly. Instead, interface automata [3] are used to specify not only the interface behaviour of components but also their assumptions about the environment. Abstracting away implementation details of components, interface automata are much simpler and easier to handle. Firstly, we ensure that each component is consistent with its corresponding interface automaton, namely, each component is able to fulfil the output guarantee of the automaton under the environmental assumptions of the automaton.

Secondly, we check the consistency of the network comprising these interface automata. This consistency can in turn justify that the process network is free from unexpected reception and deadlock. The former property indicates that the data flow between its components is directed in a way which is consistent with the assumptions made by the components. The latter refers to the absence of deadlock where the system cannot make any further progress. By adopting such a divide-and-conquer approach, the potential state space explosion problem can be alleviated.

The presented research is motivated by previous work on the Moses tool suite [4]. Moses presents an additional challenge to component-based development in that it supports the modelling and simulation of heterogeneous discrete-event systems, where components are modelled by different formalisms. For example, components can be defined [4,7,9] as process networks, Petri Nets, Statecharts, etc. The proposed approach can also largely simplify the analysis of such heterogeneous systems.

This paper is structured as follows. In Sect. 2, our approach is compared with the related work. In Sect. 3, we define discrete-event components (or processes), interface automata and the consistency between them, and present a practical method for checking the consistency. In Sect. 4, we define dataflow process networks and interface automaton networks and present our method of modular analysis of dataflow process networks. Finally, we conclude this paper in Sect. 5.

2 Related Work

Interface automata were first introduced in [3]. The authors established a simple and well-defined semantics for them and defined their composition by two-party synchronization. Also, alternating simulation was proposed to determine a refinement relationship between interface automata. This relationship takes an optimistic view of the environment by assuming that it is always helpful, only supplying inputs expected by an automaton.

This optimistic view allows more possible implementations than a pessimistic approach where the environment can behave as it pleases.

We take this optimistic view and adapt alternating simulation to define the consistency of components with interface automata, taking into account data values used in components. In order to prove properties such as deadlock freedom, an additional restriction is imposed which requires a component to produce at least one of the outputs that the corresponding interface automaton can produce. Also, a simpler method of checking the relation is presented, which does not require the construction of the Cartesian product of their state spaces as [3] does – only the reachable states in the product need to be constructed. Additionally, in contrast to the simple composition scheme in [3], we allow interface automata to be composed in many more ways reflecting how process networks can be constructed in practice.

Our adaptation and checking method of alternating simulation is inspired by [13], where a similar relation was proposed to check the conformance (or refinement) relationship between CCS models. However, this relation is more restricted than ours, as it requires that both specification and implementation models have no mixed states (where both input and output transitions can occur), while in our approach this is only required of the specification, *i.e.* interface automata. Furthermore, because of the presence of blocking outputs, models in their approach are different in nature from ours where a component is in full control of its outputs. In addition, in our approach the substitutability of heterogeneous components can be checked with the aid of interface automata.

There are some other approaches which also utilize the environmental assumptions of components for verification. In [5,6], the assumptions and the actual behaviours of components are derived from component specifications. The deadlock freedom of a system is determined by pairwise matching between the assumptions of a component and the actual behaviour of another component. However, the proposed method is incomplete and limited to one-to-one communication or synchronization.

The approach in [16] requires additional models of the environmental assumptions of components. The models are used to restrict the behaviour of the environment so that system deadlock can be discovered by detecting undesirable usage of components. However, the global state space needs to be built, which would easily lead to the state space explosion problem.

3 Consistency of Components

In this section, general reactive systems are first introduced. These are specialised as discrete-event components in Sect. 3.2 and as interface automata in Sect. 3.3. In Sect. 3.4, the consistency of discrete-event components with interface automata is defined. A practical method of consistency checking is presented in Sect. 3.5.

3.1 Reactive Transition Systems

Definition 1. *A* reactive transition system *(RTS) is defined as* $L = (s^0, S, \square, \rightarrow)$, *where*

– S *is a set (possibly infinite) of states and* $s^0 \in S$ *is the initial state;*

– $\square$ *is a set (possibly infinite) of events, consisting of three mutually disjoint sets of input events* $\square^I$*, output events* $\square^O$*, and internal events* $\square^H$*;*
– $\rightarrow \subseteq S \times \square \times S$ *is a set of transitions.*

This definition draws an explicit distinction between input, output and internal events. This is because a system has control over its internal and output events only, but no control over its input events. Instead, when an input event occurs is decided by the environment. In other words, the system cannot prevent the environment from producing an input event if it wants to do so. In the following, we let $\square^{obs} = \square^I \cup \square^O$ be a set of observable events and $\square^{ctl} = \square^O \cup \square^H$ be a set of controllable events of L.

Definition 2. *A* trace $\square$ *of a RTS* L *from* $s_1 \in S$ *is an event sequence* $e_1 e_2 \rhd\!\rhd\!\rhd e_m$ *such that* $\forall j\colon 1 \le j \le m$, $\exists (s_j \, e_j \, s_{j+1}) \in \rightarrow$. *State* s_{m+1} *is called* reachable in L (via $\square$) *if* $\square$ *is from* s^0. *Also, a trace is said to be* internal *if* $\forall j\colon 1 \le j \le m \, e_j \in \square^H$ *and to be* empty *if* $m = 0$. *An empty trace is denoted as* $\square$.

The restriction $\square \restriction E$ *of* $\square$ *on an event set* E *is an event sequence obtained by removing from* $\square$ *all events not in* E.

In the sequel, we write $s \xrightarrow{e} s'$ to denote $(s \, e \, s') \in \rightarrow$. We also write $s \Longrightarrow s'$ if s' is reachable via a (possibly empty) internal trace of L from s, and $s \overset{e}{\Longrightarrow} s'$ for $e \in \square^{obs}$ if $s \Longrightarrow s'' \wedge s'' \xrightarrow{e} s'$.

A RTS L is called *deterministic* if $\forall e \in \square \, s \, s' \, s'' \in S$, $s \xrightarrow{e} s' \wedge s \xrightarrow{e} s''$ implies $s' = s''$. It is said to be *nondeterministic* otherwise. Also, the sets of *enabled input and output events* at a state $s \in S$ are defined by $\mathfrak{en}^I(s) = \{e \in \square^I \mid \exists s' \in S \, (s \, e \, s') \in \rightarrow\}$ and $\mathfrak{en}^O(s) = \{e \in \square^O \mid \exists s' \in S \, s \overset{e}{\Longrightarrow} s'\}$, respectively. An event $e \in \square^I$ is said to be *refused* at s if $e \notin \mathfrak{en}^I(s)$. L is said to be *input-universal* if $\forall s \in S \, \mathfrak{en}^I(s) = \square^I$. Additionally, a state s is called a *terminal state* if $\nexists (s \, e \, s') \in \rightarrow$.

3.2 Discrete-Event Components

A Moses component is a discrete-event component that consumes data streams fed to its input ports, and produces data streams through its output ports. The input and output ports form a component's view of the rest of the system and decouple the outside world from the component. This separation allows the behaviour of the component to be described independently of its ultimate context. Likewise, the outside world learns about a component only from the communication through its ports. In defining components, we assume a universal set $\mathcal{U}^{port}$ of ports, and a countable universal set $\mathcal{U}^{val}$ of values of data flowing through ports.

Definition 3. *Given two disjoint finite sets of its input ports* $\square^I \subset \mathcal{U}^{port}$ *and its output ports* $\square^O \subset \mathcal{U}^{port}$*, a* discrete-event component *(DEC) is defined as an input-universal RTS* $C = (s^0 \, S \, \square \, \rightarrow)$*, where* $\square^I = \square^I \times \mathcal{U}^{val}$*,* $\square^O \subseteq \square^O \times \mathcal{U}^{val}$ *and* $\square^H$ *is a set of fresh labels for transitions with no external effect.*

In the definition, an input/output event of C is regarded as an occurrence of data flowing through an input/output port of C, while internal events of each DEC are considered to be unique to that DEC. Also, a DEC is an input-universal RTS, *i.e.* it never refuses

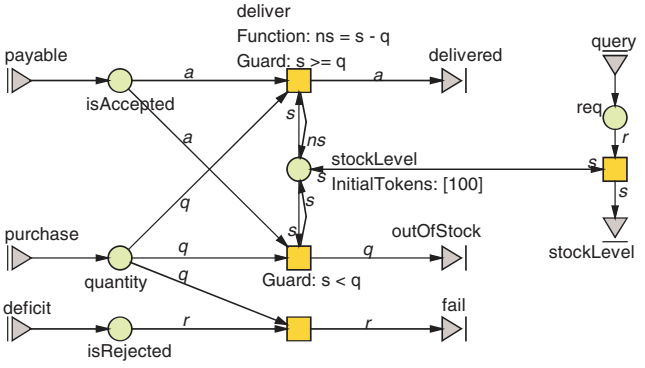


Fig. 1. A store component

an input and hence writing to a component will never block. Typically, each component has one or more input buffers (generally of infinite length), which are either implicit or explicit depending on the modelling language. For instance, a Petri Net component may have multiple places acting as explicit buffers [7], while a UML Statechart component has only one implicit buffer for all input ports [9]. These built-in buffers ensure the input acceptance of components. In the sequel, we assume a universal set $\mathcal{U}^{dec}$ of DECs and ports of each DEC to be unique to the DEC.

Figure 1 gives an example of a Petri Net component in Moses, where triangles represent the input/output ports of components and where the body of the component is given in the usual Petri Net notation with circles, boxes and arcs representing places, transitions and the flow relationships, respectively. When data comes into a component via its input port, it is added to the place(s) connected to this port. A transition (*e.g.* "deliver" in Fig. 1) becomes enabled once all its input places have enough tokens and its guard evaluates to true. As is the case for other high-level Petri nets (*e.g.* [8]), this binds the tokens to the variable names (*e.g.* "a","q" and "s") on its incoming arcs, and finally the transition fires. While firing, the transition binds the variable names (*e.g.* "ns" and "a") on the outgoing arcs depending on the values of the variables on the incoming arcs. When a firing transition is connected to an output port (*e.g.* "delivered"), data is sent out via the port to all connected components[1]. For a Petri Net component, we require input ports to be connected to places and output ports to transitions to ensure that any component output can be meaningfully connected to any component input. Assuming the interleaving semantics of Petri Net components [7], the interpretation of such components in terms of discrete-event components is straightforward and we omit this for the sake of brevity. See [14] for the basic concepts of Petri Nets, and [4,7] for more descriptions of the Moses approach to compositional Petri Nets.

This example models an online store that waits for a purchase request from a customer and payment acceptance from the customer's bank before delivering the goods. If the bank refuses to pay or the goods are out of stock, the request fails. The store also reports

[1] Ports can be considered to segment arcs into three – that part of the arc prior to the output port, that part between output and input port(s), and that part following the input ports.

the stock level when being queried. Initially, the store holds 100 pieces of goods. A successful order will result in the stock level being decreased by the ordered quantity.

3.3 Interface Automata

Usually, a component is designed under some environmental assumptions about how the component can be properly deployed, *e.g.* interaction protocols. The assumptions are useful for analysing the behaviour of the component, especially when the component is independently developed and analysed. However, an input-universal component cannot constrain the environment as to when and what kind of input to provide. Therefore, these assumptions cannot be captured by component models. In this approach we employ interface automata [3] to solve this problem.

Definition 4. *An* interface automaton[2] *(IA) is defined as a deterministic RTS* $A = (s^0, S, \square, \rightarrow)$, *where* S *and* $\square$ *are finite and* $\square^H = \emptyset$.

Like [13,17], we exclude IAs with mixed states, *i.e.* states where both input and output transitions can occur. Also, we assume a universal set $\mathcal{U}^{ia}$ of IAs.

The information captured by an interface automaton is twofold. On the one hand, the behaviour of the automaton is observed through a sequence of its output events. On the other hand, the assumption is implicitly captured that the environment should never provide an input if the automaton is in a state where the input is refused. Also, when the automaton wishes to produce an output, the environment should be ready to accept it.

As an example, suppose that we have the automaton in Fig. 2(a) as the interface specification of the store component of Fig. 1. This captures the assumption that the environment cannot provide a second purchase request before the first one has been processed. Also, after the store receives a purchase request, the environment can either provide a "payable" message indicating that the customer can pay for the purchase or a "deficit" message indicating otherwise. On the other hand, it guarantees that the store produces either a "delivered" or "outOfStock" message but definitely not a "fail" message immediately after receiving a "payable" message.

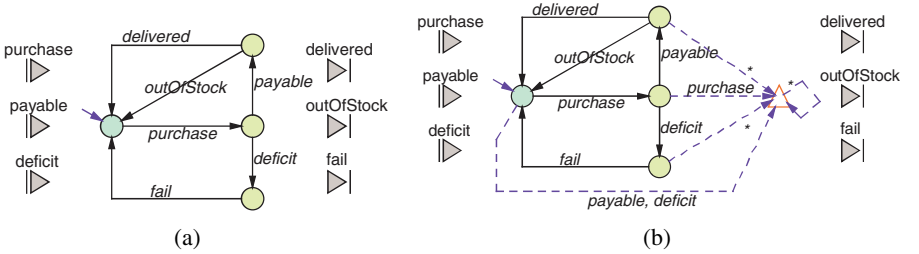


Fig. 2. A store automaton (a) and its input-universal RTS (b)

[2] Originally defined in [3], IAs can have internal events and be nondeterministic. We believe that Definition 4 is sufficiently expressive for our purpose.

Definitions 3 and 4 indicate a similarity in behaviour between interface automata and discrete-event components. Here, we consider an input event of an IA corresponds to an occurrence of data flow with an arbitrary value through an input port of a component. Similarly, an occurrence of data flow with an associated value at an output port of the component corresponds to an output event of the IA. In other words, when relating the behaviour of IAs and DECs, the events of the IAs correspond to an abstraction of the events of the DECs, an abstraction which ignores the data values. While abstracting away the implementation details of components, the high-level interface specifications can help simplify the analysis of process networks. They are also very useful in architecture analysis since component models are often not available when designing system architectures.

3.4 Consistency of Discrete-Event Components

The association of interface automata with discrete-event components leads to the most important issue in this approach, that is, the consistency of DECs with IAs. The consistency refers to the fact that an IA can safely be substituted by a DEC without compromising the properties which previously hold.

The consistency cannot be defined by traditional refinement relations, *e.g.* trace containment and simulation. This is because these only allow the implementation to have less input and output behaviour than the specification, whereas input-universal components are able to handle more inputs than IAs. Hence, we adopt alternating simulation [3] to define consistency.

Alternating simulation is concerned with the relation of an IA with a (helpful) environment. It can be considered as a two-person game, where the automaton will try to perform some action which will cause the environment to block and the environment will try to respond so that the automaton does not succeed in its attempt. Thus, the environment can limit the behaviour of the automaton by not offering certain inputs and the automaton can make things easier for the environment by not generating certain outputs. If an environment is helpful enough for an automaton, then it should also be helpful for a refinement of the automaton. The refinement can offer less outputs (since this will not make as many demands on the environment) and accept more inputs (since the environment will not offer them).

Originally, alternating simulation was used to define the refinement between two IAs, where no data values were involved [3]. We extend this refinement relation to accommodate the implementation (or DECs) with data values. Also, in order to prove properties like deadlock freedom, an additional restriction is imposed which requires the component to generate at least one of the outputs that the automaton can possibly produce.

Definition 5. *Consider an IA A and a DEC C such that $\square^I_A \subseteq \square^I_C$ and $\square^O_A \supseteq \square^O_C$. C* simulates *A, written $C \preceq A$, if there exists a relation $\preceq \subseteq S_C \times S_A$ such that $s^0_C \preceq s^0_A$ and for $q \preceq s$, the following conditions hold:*

1. $\mathbf{en}^O_A(s) \neq \emptyset$ *implies* $\mathbf{en}^O_C(q) \neq \emptyset$*;*
2. $\forall \langle f^{\frown} v\rangle \in \mathbf{en}^O_C(q) \cup (\mathbf{en}^I_A(s) \times \mathcal{U}^{val})$, $q \overset{\langle f^{\frown} v\rangle}{\Longrightarrow}_C q'$ *implies* $\exists s' \in S_A{}^{\frown} s \xrightarrow{f}_A s' \wedge q' \preceq s'$.

Basically, C simulates A if C is able to fulfil the output guarantee of A when the environment provides C only enabled inputs of A. In other words, the environment provides an input to C at a state $q \in S_C$ only when A at a state $s \in S_A$ (s.t. $q \preceq s$) is able to accept the input. Also, fulfilling the output guarantee of A indicates two facts. First, every possible output, which C can produce at q or at a state reachable via an internal trace from q, must also be allowed by A at s. Second, C from q should be able to produce at least one of the outputs that A can produce at s. The definition implies an input and output duality that C at state q allows more input events but produces less output events than A at state s. It is worth noting that $\mathfrak{en}^I_C(q) \supseteq \mathfrak{en}^I_A(s) \times \mathcal{U}^{val}$ always holds for all $q \in S_C, s \in S_A$, because C is input-universal. Note also that condition 2 implies $\mathfrak{en}^O_C(q) \subseteq \mathfrak{en}^O_A(s) \times \mathcal{U}^{val}$.

Definition 5 allows DECs with equal or less output ports to be the implementation of an IA. However, DECs often have not only more input ports but also more output ports in practice, especially when third-party components are deployed which may provide more services than needed in an application domain. To solve this, we define instantiated components for these DECs and relax the conditions of Definition 5 in defining the consistency of DECs with IAs. Note in the following definition that $\hat{C}(\mathcal{O}) = C$ if $\square^O_C \subseteq \mathcal{O}$.

Definition 6. *An* instantiated component *of a DEC C with respect to a set $\mathcal{O} \subseteq \square^O_C$ is defined by $\hat{C}(\mathcal{O}) = (s^0, S, \square_{\hat{C}}, \rightarrow_{\hat{C}})$ where $\square^I_{\hat{C}} = \square^I_C, \square^O_{\hat{C}} = \{\langle f, v\rangle \in \square^O_C \mid f \in \mathcal{O}\}$ and $\square^H_{\hat{C}} = \square^H_C \cup \square^O_C \setminus \square^O_{\hat{C}}$.*

Definition 7. *Consider an IA A and a DEC C such that $\square^I_A \subseteq \square^I_C$. C is* consistent *with A if $\hat{C}(\square^O_A) \preceq A$.*

3.5 Practical Consistency Checking of Discrete-Event Components

In this section, the method of checking consistency of DECs with IAs is presented, which utilizes the environmental assumptions captured by these IAs.

Derived Interface Automata. Before presenting the method, we need to have two auxiliary definitions – mirrors and input-universal RTSs of interface automata. The mirror of an IA A is built to represent all helpful environments with which A can be composed. A helpful environment of A is one that can always provide inputs expected by A and accept outputs generated by A. Also, any helpful environment of A should be an implementation of the mirror. In addition, we make explicit the environmental assumptions of IAs by building their input-universal RTSs, where a refused event will now be accepted but lead to an error state.

Definition 8. *Given an IA A, the* mirror *of A is an IA $M = (s^0_A, S_A, \square_M, \rightarrow_A)$ with $\square^I_M = \square^O_A$ and $\square^O_M = \square^I_A$; The* input-universal RTS *of A is a deterministic RTS $U = (s^0_A, S_A \cup \{\bot\}, \square_A, \rightarrow_U)$, where*

$$\rightarrow_U = \rightarrow_A \cup \{(\bot, f, \bot) \mid f \in \square^I_A\} \cup \{(s, f, \bot) \mid s \in S_A, f \notin \mathfrak{en}^I_A(s)\}$$

Basically, the mirror of A has the input and output events of A interchanged. Hence $\mathbf{en}^I_M(s) = \mathbf{en}^O_A(s)$ and $\mathbf{en}^O_M(s) = \mathbf{en}^I_A(s)$ hold for all $s \in S_A$. The input-universal RTS of A is constructed by adding a transition outgoing from a state $s \in S_A$ to a single error state $\perp \notin S_A$ for all refused input events at s. As an example, the input-universal RTS of Fig. 2(a) is shown in Fig. 2(b), where the white triangle "$\triangle$" represents the error state and "$*$" matches any of input events of the RTS.

Consistency Checking. In the following, a two-step method of consistency checking is presented. Firstly, the input-universal RTS of the mirror of an IA is constructed. Next the product of the component and the RTS is built. The consistency is then determined by checking in the product for the absence of error and illegal deadlock states and the possibility of continuing interactions. This is justified by Theorem 1 (below).

Definition 9. *Consider a DEC C and an input-universal RTS U such that $\square^O_U \subseteq \square^I_C$. The* product *of C and U is a RTS $L_\otimes = (s^0_\otimes, S_\otimes, \square_\otimes, \rightarrow_\otimes)$, where:*

- $s^0_\otimes = \langle s^0_C, s^0_U \rangle$ *and* $S_\otimes \subseteq S_C \times S_U$ *is the smallest set such that* $s^0_\otimes \in S_\otimes$ *and* $\forall \mathbf{s} \in S_\otimes, \mathbf{s} \xrightarrow{e}_\otimes \mathbf{s}'$ *implies* $\mathbf{s}' \in S_\otimes$.
- $\square^I_\otimes = \square^O_\otimes = \emptyset$, *and* $\square^H_\otimes = \square^{ctl}_C \cup (\square^O_U \times \mathcal{U}^{val})$;
- $\rightarrow_\otimes = \{(\langle q, s\rangle, e, \langle q', s\rangle) \mid e \in \square^{ctl}_C \setminus (\square^I_U \times \mathcal{U}^{val}), q \xrightarrow{e}_C q'\}$
  $\bigcup \{(\langle q, s\rangle, \langle f, v\rangle, \langle q', s'\rangle) \mid \langle f, v\rangle \in \square_U \times \mathcal{U}^{val}, q \xrightarrow{\langle f, v\rangle}_C q' \wedge s \xrightarrow{f}_U s'\}$

Theorem 1. *Consider a DEC C and an IA A such that $\square^I_A \subseteq \square^I_C$. Let $L_\otimes$ be the product of C and the input-universal RTS U of A's mirror. Then C is consistent with A if $\forall \langle q, s\rangle \in S_\otimes$, the following conditions hold:*

1. *s is not an error state,* i.e. *$s \neq \perp$;*
2. *If $\langle q, s\rangle$ is a terminal state, then s is a terminal state of A.*
3. *If $\langle q, s\rangle$ is not a terminal state, then $\exists s' \neq s, \langle q', s'\rangle \in S_\otimes$ such that $\langle q', s'\rangle$ is reachable from $\langle q, s\rangle$ in $L_\otimes$.*

Proof. Let $\hat{C}$ represent $\hat{C}(\square^O_A)$, $\square$ be a relation $\{\langle q, s\rangle \in S_\otimes \mid q \in S_{\hat{C}}, s \in S_A\}$, we prove $\square$ is a simulation relation between $\hat{C}$ and A by induction. First, $(s^0_{\hat{C}}, s^0_A) \in \square$. Next, suppose $(q, s) \in \square$,

1. If $\mathbf{en}^O_A(s) \neq \emptyset$, s is not a terminal state in A. Due to condition 2, $\langle q, s\rangle$ is not a terminal state in $L_\otimes$ either. Because mixed states are assumed to be absent in A, $\mathbf{en}^I_A(s) = \mathbf{en}^O_U(s) = \emptyset$. Because of condition 3, $\exists \langle q'', s\rangle \xrightarrow{\langle f, v\rangle}_\otimes \langle q', s'\rangle$ such that $q \Longrightarrow_{\hat{C}} q''$ and $f \in \square^I_U$ (note that $\square^I_U = \square^O_A$). From Definition 9, $\langle f, v\rangle \in \mathbf{en}^O_{\hat{C}}(q)$ holds, *i.e.* $\mathbf{en}^O_{\hat{C}}(q) \neq \emptyset$.
2. For $e \in \square^{ctl}_C \setminus (\square^I_U \times \mathcal{U}^{val})$, if $q \xrightarrow{e}_C q'$, then $q \xrightarrow{e}_{\hat{C}} q'$. Hence $(q', s) \in \square$;
3. For $\langle f, v\rangle \in \mathbf{en}^O_{\hat{C}}(q) \cup (\mathbf{en}^I_A(s) \times \mathcal{U}^{val})$, if $q \xrightarrow{\langle f, v\rangle}_C q'$, then $q \xrightarrow{\langle f, v\rangle}_{\hat{C}} q'$. $\exists s' \in S_U$, $s \xrightarrow{f}_U s'$ holds for $f \in \mathbf{en}^I_A(s)$. It also holds for $\langle f, v\rangle \in \mathbf{en}^O_{\hat{C}}(q)$ since $\mathbf{en}^O_{\hat{C}}(q) \subseteq \square^O_A \times \mathcal{U}^{val}$ and U is input-universal w.r.t. $\square^O_A$. From Definition 9, we can get $\langle q', s'\rangle \in S_\otimes$. Due to condition 1, $(q', s') \in \square$ holds.

Therefore, □ is a simulation relation between $\hat{C}$ and A. From Definition 7, C is consistent with A. □

In the theorem, condition 1 indicates the input and output duality between C and A. Condition 2 requires the absence of illegal deadlock states in the product. Finally, condition 3 states a requirement on the reactive nature of C, that is, C should be active in communication.

Now we are able to check the consistency of the store component of Fig. 1 with the store automaton of Fig. 2(a). We calculate the product of the component model and the input-universal RTS of the automaton's mirror and check it against the above conditions. If these conditions are satisfied, then Theorem 1 allows us to conclude that the store component is consistent with the store automaton. At the time of writing, this algorithm has been implemented in the context of Moses.

4 Modular Analysis of Component Networks

4.1 Dataflow Process Networks

There are many kinds of process networks [10,11,12,15]—they differ, e.g., in their model of communication (explicit FIFO buffers between processes vs synchronous communication), or in their model of execution (as an interleaving of atomic and non-blocking *firings* of processes vs a continuous and possibly blocking thread-like execution of each process in parallel with all other processes).

In this paper we consider the form proposed in [15]. Basically, a process network consists of a collection of concurrently executing processes with ports and a set of channels connecting the output and input ports of these processes. Often, the channels represent FIFO buffers between processes, but we consider that the buffers are encapsulated in their destination processes and the channels represent only the causality of data flow between processes. Due to the localization of buffers, the semantic definition of process networks is simplified and thus facilitates modular analysis. Furthermore, it also gives us the flexibility to model a variety of buffers thanks to the diversity of component modelling formalisms.

Definition 10. *A dataflow process network (DPN) is defined by* $D = (P, R)$*, where* $P \subset \mathcal{U}^{dec}$ *is a set of processes and* $R \subseteq \bigcup_{p \in P} □_p^O \times \bigcup_{p \in P} □_p^I$ *is a set of connections relating the output ports to the input ports of the processes, such that*[3]

- $(o, i) \in R$ *implies* $□(o) \neq □(i)$*;*
- $(o, i), (o, i') \in R \wedge i \neq i'$ *implies* $□(i) \neq □(i')$*;*

where $□(f) = p$ *if* $p \in P$ *and* $f \in □_p^I \cup □_p^O$.

[3] The function $\rho(f)$ returns the process associated with port f. Here, we exclude the situations where R connects an output port and an input port of one process and where more than one connection originating from one output port ends at two or more input ports of a process. This is because these introduce true concurrency at component boundaries, which in turn contradicts the interleaving semantics of interface automata. This will be addressed in future work.

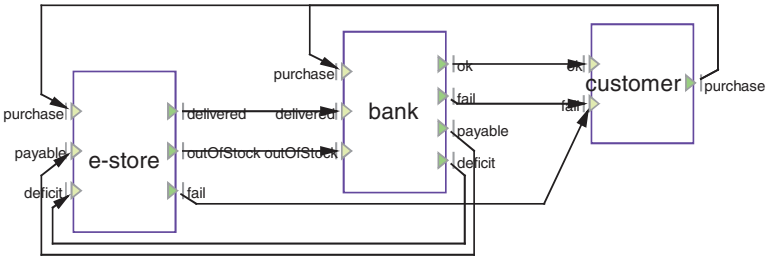


Fig. 3. An online purchase DPN

A DPN can be depicted as a directed graph. At this level of abstraction, each node represents a process, each triangle associated with a node represents an input/output port of the node, and each edge represents a connection between ports. Figure 3 shows an example: an online purchase DPN. When a customer sends a purchase request, the request goes simultaneously to the store and the bank. These then collaborate to process the request and finally report back to the customer whether the purchase succeeds or fails (detail will be given later).

A variety of communication structures are supported by DPNs where data can be relayed, duplicated, and merged among processes. For example, two connections starting from the port "purchase" of the customer and the connections ending at the port "fail" of the customer in Fig. 3 demonstrate the last two situations, respectively. Furthermore, disconnected input and output ports of processes in a DPN are allowed. A disconnected input port will receive no data, while a disconnected output port will discard all data sent to it. We further define the sets of *connected input and output ports* of a process $p \in P$ in a DPN D as $\square_p^I = \{i \in \square_p^I \mid (o, i) \in R\}$ and $\square_p^O = \{o \in \square_p^O \mid (o, i) \in R\}$, respectively.

Definition 11. *Consider a DPN $D = (P, R)$. Let $R^\square = \{(\langle o, v\rangle, \langle i, v\rangle) \mid (o, i) \in R, v \in \mathcal{U}^{val}\}$ be a causality relation between output and input events. Then the* product *of D is defined as a RTS $L_D = (s^0, S, \square, \rightarrow_D)$, where*

- $s^0 = \square_{p\in P} s_p^0$ *and* $S \subseteq \square_{p\in P} S_p$. *We let projections* $\square_p : S \to S_p$ *and let* $\mathbf{s}_p = \square_p(\mathbf{s})$ *and* $\mathbf{s}'_p = \square_p(\mathbf{s}')$ *for* $p \in P, \mathbf{s}, \mathbf{s}' \in S$;
- $\square^I = \square^O = \emptyset$ *and* $\square^H = \bigcup_{p\in P} \square_p^{ctl}$;
- $\rightarrow_D = \{(\mathbf{s}, e, \mathbf{s}') \mid e \in \square_p^{ctl}, \mathbf{s}_p \xrightarrow{e}_p \mathbf{s}'_p \wedge \forall g \in P, g \neq p \wedge \mathbf{s}'_g = \square(\mathbf{s}_g, p, e)\}$

where $\square(\mathbf{s}_g, p, e) = \begin{cases} q & \text{if } e \in \square_p^O \wedge \exists e' \in \square_g^I, q \in S_g, (e, e') \in R^\square \wedge \mathbf{s}_g \xrightarrow{e'}_g q \\ \mathbf{s}_g & \text{otherwise} \end{cases}$.

The product of a DPN captures the semantics of the DPN. According to the definition, a state of a DPN is a vector of states of all its processes, and a DPN transits between states by simply executing one of its processes and directing data flow, if any, according to the connections R. A transition of the DPN is either an internal transition or an output transition of a process. The latter may involve synchronous execution of multiple input transitions of other processes, depending on R.

In order to facilitate further analysis, we define projections of traces of DPNs, which relates the behaviour of a DPN to that of its processes.

Definition 12. *Given a DPN $D = (P, R)$ and a trace σ of L_D from s_D^0, the* trace projection *of σ on $p \in P$ is a trace of p from s_p^0, denoted as $\sigma_p(\sigma)$, obtained by first removing from σ all events not in $\sigma_p^{ctl} \cup X$ and then renaming all events $x \in X$ to $y \in \sigma_p^I$ s.t. $(x, y) \in R^\sigma$, where $X = \{x \mid \exists y \in \sigma_p^I . (x, y) \in R^\sigma\}$.*

4.2 Interface Automaton Networks

We define the composition of the IAs by interface automaton networks as for DECs.

Definition 13. *An* interface automaton network *(IAN) is defined as $N = (W, R)$, where $W \subset \mathcal{U}^{ia}$ and $R \subseteq \bigcup_{a \in W} \sigma_a^O \times \bigcup_{a \in W} \sigma_a^I$ is a causality relation between the output and input events of the IAs, such that*[4]*:*

- *$(o, i) \in R$ implies $\sigma(o) \neq \sigma(i)$;*
- *$(o, i), (o, i') \in R \wedge i \neq i'$ implies $\sigma(i) \neq \sigma(i')$;*

where $\sigma(f) = a$ if $a \in W$ and $f \in \sigma_a$.

The semantics of IANs is captured by their products defined below.

Definition 14. *Consider an IAN $N = (W, R)$. Let B be the set of input-universal RTSs of all IAs in W. Then the* product *of N is defined as a RTS $L_N = (s^0, S, \sigma, \rightarrow_N)$, where:*

- *$s^0 = \sigma_{b \in B} s_b^0$ and $S \subseteq \sigma_{b \in B} S_b$ is the smallest set such that $s^0 \in S$ and $\forall \mathbf{s} \in S, \mathbf{s} \xrightarrow{f}_N \mathbf{s}'$ implies $\mathbf{s}' \in S$. We let projections $\sigma_b : S \rightarrow S_b$ and let $\mathbf{s}_b = \sigma_b(\mathbf{s})$ and $\mathbf{s}'_b = \sigma_b(\mathbf{s}')$ for $b \in B, \mathbf{s}, \mathbf{s}' \in S$;*
- *$\sigma^I = \sigma^O = \emptyset$, and $\sigma^H = \bigcup_{b \in B} \sigma_b^O$;*
- *$\rightarrow_N = \{(\mathbf{s}, f, \mathbf{s}') \mid f \in \sigma_b^O, \mathbf{s}_b \xrightarrow{f}_b \mathbf{s}'_b \wedge \forall h \in B, h \neq b \wedge \mathbf{s}'_h = \sigma(\mathbf{s}_h, f)\}$,*
  where $\sigma(\mathbf{s}_h, f) = \begin{cases} q & \text{if } \exists i \in \sigma_h^I, q \in S_h, (f, i) \in R \wedge \mathbf{s}_h \xrightarrow{i}_h q \\ \mathbf{s}_h & \text{otherwise} \end{cases}$.

As an example, suppose that we have an IAN where W consists of the interface automata of Figs. 2(a), 4(a) and 4(b), and R defines their composition as shown in Fig. 3. Then the product of the IAN is shown in Fig. 5.

Definition 15. *Consider an IAN $N = (W, R)$. Let L_N be the product of N. Then N is* consistent *if no error or deadlock states are reachable in L_N,* i.e. *$\forall \mathbf{s} \in S_N$, the following conditions hold:*

1. *$\forall a \in W, \sigma_a(\mathbf{s}) \neq \perp$;*
2. *If $\mathbf{s}$ is a terminal state of L_N, then $\forall a \in W, \mathfrak{en}_a^I(\sigma_a(\mathbf{s})) = \emptyset$.*

[4] These well-formedness rules are introduced for the same reason as in Definition 10.

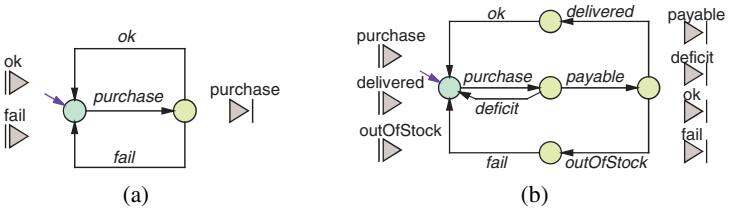


Fig. 4. The customer automaton (a) and the bank automaton (b)

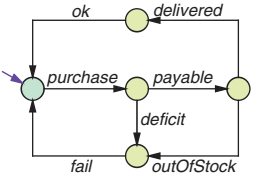


Fig. 5. The product of the example IAN

Basically, the consistency of IANs ensures the absence of mismatches of environmental assumptions and output guarantees of processes and also the freedom of deadlock at a high level of abstraction. For example, as the product state space shown in Fig. 5 contains no error or deadlock state, the IAN is consistent.

Since IAs can specify the interface behaviour of DECs, IANs can capture the interaction behaviour of DECs in DPNs. Hence, as we shall see, the consistency of IANs can serve as the basis of analysis of DPNs. Also, it is cheaper to determine the consistency of IANs, because IANs generally have smaller state space than DPNs.

4.3 Properties of Dataflow Process Networks

In this section, we define the properties of DPNs such as safety and deadlock freedom. The safety considered here refers to the fact that the environmental assumptions made by processes are respected in executions of DPNs. More specifically, no unexpected reception of data at any input port ever occurs. Additionally, a DPN is said to be deadlocked if it reaches a state where no process can make any progress, generally because each is blocked waiting for an input from others, while the event cannot occur. Deadlock freedom refers to the ability of DPNs to make progress or perform computations.

Definition 16. *A dataflow process network D is* sketched by *a total function $\mathcal{A}: P \rightarrow \mathcal{U}^{ia}$ if $\forall (p, a) \in \mathcal{A} \cdot I_a = I_p \cdot O_a = O_p$ and p is consistent with a, where I_p denotes the connected ports of p in D as defined in Sect. 4.1.*

Consider a DPN D sketched by a total function $\mathcal{A}$. Let L_D be the product of D, σ be a trace of L_D from s_D^0, $p \in P$, $a = \mathcal{A}(p)$, $\hat{p}$ represent $\hat{p}(O_p)$, $\sigma_p = \sigma_p(\sigma) \upharpoonright \Sigma_p^{obs}$ be the observable sequence of the trace projection of σ on p, and σ_a be the sequence of ports involved in σ_p (Note $|\sigma_p| = |\sigma_a|$). Then we can formulate properties of D in Definitions 17 and 18 below.

Definition 17. *A trace* $\square$ *is* free of unexpected reception *in* D *if* $\square_a$ *is a trace of* a *from* s_a^0 *for all* $p \in P$. D *is* free of unexpected reception *if all traces of* L_D *from* s_D^0 *are free of unexpected reception.*

Lemma 1. *Consider a trace* $\square$ *which is free of unexpected reception. Let* $\mathbf{q} \in S_D$ *be a reachable state via* $\square$ *in* L_D *and* $\mathbf{s}_a$ *be a reachable state via* $\square_a$ *in* a*, then (1)* $\mathbf{s}_a$ *is the only state reachable via* $\square_a$ *in* a *and (2)* $\square_p(\mathbf{q}) \preceq \mathbf{s}_a$.

Proof. (1) holds because $\square \downarrow_a^H = \emptyset$ and a is assumed to be deterministic. Also, because every event f in $\square_a$ corresponds an event $\langle f, v\rangle$ in $\square_p$, (2) holds from Definition 7. $\square$

Definition 18. *Consider a DPN* D *which is free of unexpected reception. Let* $\mathbf{q} \in S_D$ *be a reachable state via* $\square$ *in* L_D *and* $\mathbf{s}_a$ *be the reachable state via* $\square_a$ *in* a *for all* $p \in P$. $\mathbf{q}$ *is a* deadlock state *if it is a terminal state in* L_D *and* $\exists p \in P, \mathfrak{en}_a^I(\mathbf{s}_a) \neq \emptyset$. D *is* free of deadlock *if no deadlock state is reachable via any trace from* s_D^0.

4.4 Property Deduction

Theorem 2. *A DPN* $D = (P, R)$ *is free of unexpected reception and deadlock if there exist both a total function* $\mathcal{A}\colon P \to \mathcal{U}^{ia}$ *s.t.* D *is sketched by* $\mathcal{A}$ *and also a consistent IAN* $N = (W, R)$*, where* $W = \{\mathcal{A}(p) \mid p \in P\}$.

Proof. We prove this theorem by induction over the length of any trace $\square$ from s_D^0. Let $\mathbf{q} \in S_D$ be a reachable state via $\square$, $p \in P$, $a = \mathcal{A}(p)$, $\hat{p}$ represent $\hat{p}(\square \downarrow_p^O)$, $\square_p = \square_p(\square) \restriction \square \downarrow_{\hat{p}}^{obs}$ and $\square_a$ be the sequence of ports involved in $\square_p$. At each step, we prove that (1) $\square$ is free of unexpected reception; (2) $\exists \mathbf{s} \in S_N, \forall a, \square_a(\mathbf{s})$ is the reachable state via $\square_a$ in a; and (3) $\mathbf{q}$ is not a deadlock state.

1. If $\square = \square$, then $\mathbf{q} = s_D^0$. Clearly, (1) holds. Let $\mathbf{s} = s_N^0$, then (2) holds. Suppose that $\mathbf{q}$ is a terminal state in L_N, *i.e.* $\forall p, \nexists e \in \square \downarrow_p^{ctl}, (\square_p(\mathbf{q}), e, q) \in \rightarrow_p$. Hence, $\forall p, \mathfrak{en}_{\hat{p}}^O(\square_p(\mathbf{q})) = \emptyset$. Because $\square_p(\mathbf{q}) \preceq \square_a(\mathbf{s})$, $\forall a \in W, \mathfrak{en}_a^O(\square_a(\mathbf{s})) = \emptyset$ and thus $\mathbf{s}$ is a terminal state. Because $\mathbf{s} \in S_N$ and thus $\mathbf{s}$ is not a deadlock state, we have $\forall a, \mathfrak{en}_a^I(\square_a(\mathbf{s})) = \emptyset$. Therefore, (3) holds.
2. Suppose $\square = e_1 e_2 \bowtie\bowtie e_m$ *s.t.* (1-3) hold on $\square$. Given a trace $\square' = \square \cdot e$, we shall prove (1-3) hold on $\square'$. Let $\mathbf{s} \in S_N$ be the state satisfying (2), $\mathbf{q}_p = \square_p(\mathbf{q})$ and $\mathbf{s}_a = \square_a(\mathbf{s})$ for all $p \in P$, and $\square'_p$ and $\square'_a$ are defined over $\square'$. Then from Lemma 1 we have $\forall p, \mathbf{q}_p \preceq \mathbf{s}_a$.
   a) if $e \in \square \downarrow_p^{ctl} \setminus (\square \downarrow_a^O \times \mathcal{U}^{val})$, then $\square'_a = \square_a$ and (1) holds. Let $\mathbf{s}' = \mathbf{s}$, then $\mathbf{s}' \in S_N$ and $\mathbf{s}'$ is reachable via $\square'$. Thus (2) holds. Same as item 1, we can prove (3).
   b) if $e \in \square \downarrow_a^O \times \mathcal{U}^{val}$ and $\mathbf{q}_p \xrightarrow{e}_p \mathbf{q}'_p$, let $e = \langle f, v\rangle$, then $\exists \mathbf{s}'_a \in S_a, \mathbf{s}_a \xrightarrow{f}_a \mathbf{s}'_a \wedge \mathbf{q}'_p \preceq \mathbf{s}'_a$ (because $\mathbf{q}_p \preceq \mathbf{s}_a$). Thus $\square'_a = \square_a \cdot f$ is a trace of a. For $g \in P \wedge g \neq p$, we let $h = \mathcal{A}(g)$. Then,
      i. if $\exists \langle f', v\rangle \in \square \downarrow_g^I, (f, f') \in R$, then $\exists \mathbf{q}'_g \in S_g, \mathbf{q}_g \xrightarrow{\langle f', v\rangle}_g \mathbf{q}'_g$. Since no error state exists in L_N (def. 15), $f' \in \mathfrak{en}_h^I(\mathbf{s}_h)$ and thus $\exists \mathbf{s}'_h \in S_h, \mathbf{s}_h \xrightarrow{f'}_h \mathbf{s}'_h$ such that $\mathbf{q}'_g \preceq \mathbf{s}'_h$. Hence $\square'_h = \square_h \cdot \langle f', v\rangle$ is a trace of h.
      ii. otherwise, $\square'_h = \square_h$ is a trace of h.

Therefore, (1) holds on $\square'$. From def. 14, $\exists s' \in S_N, \forall a \in \square_a(s') = s'_a$ and s'_a is reachable via $\square'_a$. Hence, (2) holds on $\square'$. (3) can be proved on $\square'$ as in item 1.

Therefore, the theorem holds. □

With this theorem, we can conclude the example process network of Fig. 3 is free of unexpected reception and deadlock, provided that the concrete component models of the bank and the customer are consistent with their corresponding interface automata, respectively.

In the context of Moses, we have also implemented the check for consistency of IANs. This, together with the check based on Theorem 1, gives us the ability to analyse DPNs for properties such as freedom from deadlock and unexpected reception.

5 Conclusion

In this paper a modular analysis method for dataflow process networks has been presented, where interface automata are associated with processes (or components) to specify both their interface behaviour requirements and possible environmental assumptions. Based on the IAs, we deduce the properties of DPNs, such as freedom from unexpected reception and deadlock, by checking the consistency of components and of interface automaton networks. As these checks only need to handle smaller state spaces than the traditional single monolithic check, the state space explosion problem can be alleviated. At this stage, we have implemented the algorithms for checking these two kinds of consistency in the Moses tool suite, with the development of a visual notation for interface automata and tools for their composition and compatibility checking.

The introduced interface automata can specify the behaviour of components at a high level of abstraction and serve as the contracts between architecture designers and component developers. In this way, highly independent development of components and the communication structure among components is supported. Also, this acknowledges that a system is usually designed with assumptions made about the abstract behaviour of components and that components are designed assuming particular interaction patterns with their environment.

In addition, the proposed method simplifies substitutability checking between heterogeneous components using an intermediate interface automaton. That is to say, a component can be substituted by another component in a process network if they are both consistent with the same interface automaton. Hence, the evolution of systems is supported both at the abstract level by the substitutability of interface automata and also at the component level by the substitutability of components.

The research presented here is a step towards the automated consistency checking of heterogeneous systems where system components as well as system architectures are potentially expressed in different description languages. We are investigating the application of this method to architectural models described in other languages such as Petri Nets. Currently, the assumptions of components on data values are not captured in this method. A possible way to improve this is to enhance the formalism of interface automata to support data values on input and output events. Furthermore, true concurrency at component boundaries is not considered here and will be the subject of future work.

References

1. M. Anlauff, P. Kutter, A. Pierantonio, and A. Sünbül. Using domain-specific languages for the realization of component composition. In *Proc. of the Fundamental Approaches to Software Engineering (FASE 2000)*, LNCS 1783. Springer.
2. F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau. Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008, 2000.
3. L. de Alfaro and T. Henzinger. Interface automata. In *Proc. of the Foundation of Software Engeneering (FSE 2001)*, pages 109–122. ACM Press.
4. R. Esser and J. Janneck. Moses - a tool suite for visual modelling of discrete-event systems. In *Symposium on Visual/Multimedia Approaches to Programming and Software Engineering*, 2001.
5. P. Inverardi and S. Uchitel. Proving deadlock freedom in component-based programming. In *Proc. of the Fundamental Approaches to Software Engineering (FASE 2001)*, LNCS 2029.
6. P. Inverardi, A. Wolf, and D. Yankelevich. Static checking of system behaviors using derived component assumptions. *ACM Trans. on Software Engineering and Methodology*, 9(3):239–272, 2000.
7. J. Janneck and R. Esser. Higher-order Petri Net modeling–techniques and applications. In *Workshop on Softw. Eng. and Formal Methods of ICATPN 2002*.
8. K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *Monographs in Theoretical Computer Science*. Springer-Verlag, 1997.
9. Y. Jin, R. Esser, and J. Janneck. Describing the syntax and semantics of UML statecharts in a heterogeneous modelling environment. In *Proc. of the Diagrammatic Representation and Inference (Diagrams 2002)*, LNAI 2317. Springer.
10. G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 1974*, pages 471–475. North-Holland Publishing Co.
11. R. Karp and R. Miller. Properties of a model for parallel computations: determinacy, termination, queuing. *SIAM J. Appl. Math.*, 14:1390–1411, 1966.
12. E. Lee and T. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.
13. S. Rajamani and J. Rehof. Conformance checking for models of asynchronous message passing software. In *Proc. of the Computer-Aided Verification (CAV 2002)*, LNCS 2404. Springer.
14. W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, Advances in Petri Nets, LNCS 1491. Springer-Verlag, 1998.
15. D. Skillcorn. Stream languages and data-flow. In *Advanced Topics in Data-Flow Computing*. Prentice Hall, 1991.
16. S. Uchitel and D. Yankelevich. Enhancing architectural mismatch detection with assumptions. In *Proc. of the Eng. of Computer Based Systems (ECBS 2000)*.
17. D. Yellin and R. Storm. Protocol specifications and component adaptors. *ACM Trans. on Programming Languages and Systems*, 19(2):292–333, 1997.

Foundations of a Weak Measurement-Theoretic Approach to Software Measurement

Sandro Morasca

Dipartimento di Scienze Chimiche, Fisiche e Matematiche
Università degli Studi dell'Insubria, Via Valleggio 11
I-22100, Como, Italy
sandro.morasca@uninsubria.it

Abstract. Measurement Theory has been proposed as an approach for providing software measurement with a sound basis. However, Measurement Theory has hardly ever been used for building new measures. Much more often, it has been used to analyze the properties of existing measures, but there has been little agreement on the compliance of existing measures with Measurement Theory's strict requirements. This paper introduces a modified version of Measurement Theory, called Weak Measurement Theory. Because it has weaker requirements than Measurement Theory, Weak Measurement Theory may be better suited for the needs of the state of the art of software measurement. We provide an extension of the theory of the levels of measurement and we focus on ordinal scales and extensive measurement for Weak Measurement Theory. In addition, we show how Weak Measurement Theory can be used for widening the application scope of mean values.

1 Introduction

Theoretical validation is a fundamental step that should be carried out when a measure is proposed to show (or at least provide compelling evidence) that the measure really quantifies what it purports to measure. Especially in the early years of software measurement, measures have often been defined without adequate theoretical validation, if any. The measures themselves were operational definitions of the attributes to quantify. Thus, a large number of measures were defined, but little agreement existed as to which ones really captured the software attributes they intended to measure, as little agreement existed on the properties of software attributes and of their measures.

This lack of agreement is mainly due to two reasons. First, software engineering is a relatively recent discipline, if compared to more consolidated scientific and engineering disciplines. Second, software is not a physical object, so its attributes are more "elusive" than the attributes of physical objects encounteed in everyday life. Measurement Theory (MT, for short) [3, 5], originally defined for measurement in the Social Sciences, has been proposed as an approach to making the properties of software attributes and measures explicit. MT uses a full-fledged mathematical approach and thus allows software measurers to state the properties of software attributes unambi

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 200–215, 2003.

guously. This should facilitate (1) discussing properties that each software attribute should have, (2) reaching a consensus on the properties of each software attribute, and (3) defining software measures that are consistent with intuition.

Unfortunately, MT has hardly ever been used for defining new software measures, or even for stating the properties of software attributes. To the best of this author's knowledge, no measurement-theoretic proposals exist for important software attributes such as cohesion and coupling. It is true that MT proposals exist for the properties of software complexity and size. The proposals for software complexity have been widely discussed, but the software measurement community is far from an agreement. The proposals for software size are actually inspired by the properties of the size of physical objects. As a result, MT is often viewed as a rather theoretical approach whose benefits to software measurement are quite limited.

Also, MT's requirements may be too strict for the current state of the art of the software discipline; this is probably why MT has not produced many results in software measurement or played a more important role in it. This paper provides the foundations for modifying MT to make its requirements weaker and probably better applicable to the measurement of software attributes. Thus, we introduce a modified version of MT that we call Weak Measurement Theory (WMT, for short). In addition, we provide a weakened characterization of the levels of measurement for WMT and we focus on ordinal scales and extensive measurement. Finally, we show that the mean value of a distribution can be meaningfully used even for measures defined according to WMT and ordinal measures defined according to MT.

A weaker approach such as WMT may allow us to rigorously model several software attributes in a way that does not — at least currently — fit in the framework of MT. Thus, WMT may provide several existing, currently used measures with sounder mathematical bases than they have and show their properties. Especially for elusive attributes as the ones encountered in software engineering, the definition of a measure may require several refinement steps. Modeling the properties of a software engineering attribute is hardly ever a straightforward process, in which the "right" properties of the attribute are known from the start, but a number of iterations may be required. Starting from an initial set of properties of an attribute, one or more measures may be defined. The characteristics of these measures (e.g., the ordering of the entities that these measures provide) may be used to refine the initial properties of the attribute. In turn, this leads to the definition of new measures or the refinement of existing ones. The iterations continue until a satisfactory set of properties for the software engineering attribute and a satisfactory set of measures are obtained. For many software engineering attributes, a satisfactory set of properties has not been reached yet. Thus, we propose WMT to support this refinement process so we do not reject measures that may be useful, but need refining. In addition, for some software attributes, it is unclear whether a satisfactory set of properties will eventually be obtained. WMT may be used in those cases as well.

Weakening the requirements of MT also comes with a price. WMT is less strict than MT, so there may be a loss of accuracy in the information provided by a measure that complies with our modified version. On the other hand, we obtain a theory that is more easily and naturally applicable to software measurement.

The remainder of the paper is organized as follows. Section 2 provides a critical analysis of MT. Section 3 contains the proposal of WMT and its levels of measurement. Section 4 shows two existence theorems for ordinal and extensive measurement. Section 5 show how the notion of mean can be extended. Conclusions and future research directions are in Sect. 6.

2 An Analysis of Measurement Theory

We provide an introduction to MT in Section 2.1 and point out some of its characteristics that may have hindered its application in software measurement (Section 2.2).

2.1 An Introduction to Measurement Theory

Representational Measurement Theory [3, 5] separates the "intuitive," empirical knowledge on a specified attribute of a specified set of entities, captured via the so-called Empirical Relational System (Definition 1), and the "quantitative," numerical knowledge about the attribute, captured via the so-called Numerical Relational System (Definition 2). A scale (Definition 5) maps the Empirical Relational System into the Numerical Relational System in such a way that no inconsistencies are possible.

Definition 1. Empirical Relational System. Given an attribute, let

- E denote the set of entities for which we would like to measure the attribute,
- R_1, ..., R_n denote n empirical relations capturing our intuitive knowledge on the attribute: each R_i has an *arity* n_i, so $R_i \subseteq E^{ni}$ (for notational convenience, we write either $<e_1,..,e_{ni}> \in R_i$ or $R_i(e_1,..,e_{ni})$ to denote that tuple $<e_1,..,e_{ni}>$ is in relation R_i),
- o_1, ..., o_m denote m empirical binary operations on the entities that describe how the combination of two entities yields another entity, so each o_j has the form o_j: $E \times E \rightarrow E$; these operations are represented with an infix notation, e.g., $e_3 = e_1\ o_j\ e_2$.

An Empirical Relational System is an ordered tuple $ERS=<E,R_1,...,R_n,o_1,...,o_m>$.

For example, suppose we study the control-flow complexity (attribute) of program segments (set of entities), one of the most studied attributes. We characterize it via the empirical binary relation *more_complex_than* $\subseteq E \times E$, i.e., we have $<e_1,e_2> \in$ *more_complex_than* if we believe that e_1 is more complex than e_2. The operation $\oplus$ may be the concatenation operation, i.e., $e_3 = e_1 \oplus e_2$.

The Empirical Relational System does not make use of numbers or any kind of measurement values, which are introduced by the Numerical Relational System.

Definition 2. Numerical Relational System. Let

- V denote a set of values,
- S_1, ..., S_n be n relations on the values, each S_i has the same *arity* of R_i, so $S_i \subseteq V^{ni}$,

- $\bullet_1$, ..., $\bullet_m$ denote m numerical binary operations: each operation $\bullet_j$ is of the form $\bullet_j$: $V \times V \rightarrow V$, and is usually represented with an infix notation, e.g., $v_3 = v_1 \bullet_j v_2$.

A Numerical Relational System is an ordered tuple $NRS = <V, S_1, ..., S_n, \bullet_1, ..., \bullet_m>$.

In our program complexity example, V may be the set of nonnegative integer numbers, a binary relation for may be >, and a binary operation may be — *for instance* — the sum between two integer values. The Numerical Relational System in itself does not provide any information about the entities and the attribute.

The Empirical Relational System and the Numerical Relational System are linked via a measure (Definition 3), which associates entities and values, and a scale (Definition 5), which associates the elements of the tuple of the Empirical Relational System with elements of the Numerical Relational System.

Definition 3. Measure. A measure is a function $m: E \rightarrow V$.

A measure is only a correspondence between entities and values, but does not take into account the empirical knowledge on the attribute (expressed through the empirical relational system) and how it is translated into numerical knowledge (expressed through the numerical relational system). Thus, not all measures built according to Definition 3 are sensible ones. For instance, given three program segments e_1, e_2, e_3 such that $<e_1,e_2> \in$ *more_complex_than* and $<e_2,e_3> \in$ *more_complex_than*, a measure m may be such that $m(e_1) > m(e_2)$ and $m(e_3) > m(e_2)$. To quantify an attribute sensibly, a measure must be consistent with the empirical knowledge about the attribute, i.e., a measure must satisfy the following representation condition.

Definition 4. Representation Condition. A measure must satisfy the two conditions

$$\forall i \in 1..n \; \forall <e_1, .., e_{n_i}> \in E^{n_i} \; (<e_1, .., e_{n_i}> \in R_i \Leftrightarrow <m(e_1), .., m(e_{n_i})> \in S_i) \qquad (1)$$

$$\forall j \in 1..m \; \forall <e_1, e_2> \in E \times E \; (m(e_1 \; o_j \; e_2) = m(e_1) \bullet_j m(e_2))$$

In our complexity-related example, the Representation Condition states that, $<e_1,e_2> \in$ *more_complex_than* for any two program segments e_1, e_2 iff $m(e_1) > m(e_2)$ and $m(e_1 \oplus e_2) = m(e_1) + m(e_2)$. The above definitions lead to the concept of a scale.

Definition 5. Scale. A scale is a triple $<ERS, NRS, m>$, where *ERS* is an Empirical Relational System, *NRS* is a Numerical Relational System, and m is a measure that satisfies the Representation Condition.

In what follows, we assume that measures satisfy the Representation Condition.

Given an Empirical Relational System and a Numerical Relational System, two issues arise, i.e., existence and uniqueness of a scale that links them. We will not deal with the existence problem in this section. As for uniqueness, more than one legitimate scale may be built. This is a well known fact in everyday life, in which one may measure the weight of objects in kilograms, grams, pounds, ounces, etc. At any rate, some properties of scales are invariant, called meaningful statements.

Definition 6. Meaningful Statement. A statement is said to be meaningful if its truth value does not change when a scale is replaced by another scale. Formally, if $S(m)$ is a statement based on measure m and $S(m')$ is the same statement obtained by replacing m with m', we have $S(m) \Leftrightarrow S(m')$.

Meaningful statements provide the real information content of a scale. For instance, it makes sense to say that an object is twice as heavy as another, regardless of the scale used (kilograms, pounds, etc.). If this statement is true with one scale, it is also true with all other scales. Instead, suppose we can tell if a software failure is more severe than another and we can classify failures with a 4-value severity scale with values 1 (least severe), 2, 3, 4 (most severe). It does not make sense to say that severity 2 failures are twice as severe as severity 1 failures, as the truth value of this statement depends on the specific choice of values. The truth value of the statement changes with another scale with values, say, 2, 15, 34, 981.

As the weight example shows, it may be possible to map one scale into another. In the weight example, we can map one scale into another by multiplication by a suitable constant, i.e., any proportional transformation provides a legitimate scale with a different weight unit. This leads to the definition of admissible transformation.

Definition 7. Admissible Transformation. Given a scale $<ERS,NRS,m>$, the transformation of scale f is admissible if $m' = f \circ m$ (i.e., m' is the composition of f and m) and $<ERS,NRS,m'>$ is a scale.

An admissible transformation may not always exists. For instance (see [5]), let $ERS = <E,R>$, with $E = \{r,s,t\}$ and $R = \{<r,s>,<r,t>\}$, and let $FRS = <Real, >_1>$, where *Real* is the set of real numbers and $>_1$ and $>_1$ is a binary relation on *Real* such that $x >_1 y$ if and only if $x > y+1$. Let m' be such that $m'(r) = 2$, $m'(s) = 0$, $m'(t) = 0$, and m'' such that $m''(r) = 2$, $m''(s) = 0.1$, $m''(t) = 0$. Both m' and m'' are legitimate scales, but there is no admissible transformation that transforms m' into m''. Scales of this kind do exist in MT, where they are called irregular scales.

The scales for which admissible transformations exist may be classified according to the set of admissible transformations they can undergo, since a specific set of admissible transformations entails a specific set of meaningful statements. Next we describe some basic scale types of MT (though other types exist as well).

Nominal Scales. The values of these scales are labels for categories in which the entities are classified, with *no notion of order* among the categories, i.e., the set of entities is partitioned in a set of equivalence classes, each associated with a value of the measure. These scales can be transformed into other scales through one-to-one transformations. The invariant property is that two entities belong to the same equivalence class according to some measure m if and only if they belong to the same equivalence class according to some other measure m'. An example of a nominal scale may be the programming language used to write a program.

Ordinal Scales. The values of these scales are labels of categories in which the entities are classified, with a *total ordering* across categories. These scales are transformed into other scales via strictly monotonic transformations, which preserve the invariant property of the ordering. An example of an ordinal scale is failure severity.

Interval Scales. Each entity is associated with a numerical value, so that we quantify the difference between values. Strictly speaking, given a scale m, a new scale m' can be obtained only through transformations of the kind $m'=am+b$, with $a>0$, i.e., we can change the origin of the values (by changing b) and the unit measure (by changing a). Given four entities e_1, e_2, e_3, e_4, the invariant statements are of the kind $(m(e_1) - m(e_2))/(m(e_3) - m(e_4))$: the ratio of the lengths of two intervals is invariant. An interval scale may be the date of a milestone of a software project.

Ratio Scales. Each entity is associated with a numerical value that allows us to provide a quantification for the ratio between values. Given a scale m, a new scale m' can be obtained only through transformations of the kind $m'=am$, with $a>0$, i.e., we can change only the unit measure by changing a. Given two entities e_1, e_2 the invariant statements are of the kind $m(e_1)/m(e_2)$, i.e., the ratio of the values is invariant. An example of a ratio scale may be LOC as a measure of the size of a program.

The larger the set of admissible transformations, the less "precise" the scale. To measure the control-flow complexity of program segments, a nominal scale would allow us to classify segments into different classes, but not to order them. An ordinal scale would allow us to order the classes of the program segments, but not to measure the "distance" in control-flow complexity. An interval scale would allow us to measure the "distance" in control flow complexity among the program segments, but not the ratios between control-flow complexities; a ratio scale would allow us to measure the ratios between control-flow complexities. In practical use, there is a divide between the fully numerical scales (interval, and ratio ones) and the scales whose values may not necessarily be numerical (nominal and ordinal ones).

2.2 Issues in Applying MT in Software Engineering Measurement

Representational Measurement Theory is a consistent theory that has been developed in the last decades to give a solid mathematical basis to measurement for the social sciences. Here, we show some problems that may arise when applying MT in software engineering measurement. We argue that MT may be too demanding at this stage of technology, so a weaker approach may be more appropriate.

The problem of finding a measure that satisfies the Representation Condition (even when the Representation Condition is not fully explicitly stated) has been largely debated for a number of software attributes (e.g., size, complexity, cohesion, coupling, ...) in several different ways in the literature. The main point of most of these discussions is that, for virtually any measure, a "counterexample" can be found that questions the measure's consistency with intuition, i.e., more rigorously, its satisfaction of the Representation Condition.

The first part of the Representation Condition (marked with (1)) requires that $\forall i \in 1..n \; \forall <e_1, .., e_{n_i}> \in E^{n_i} \; (<e_1, .., e_{n_i}> \in R_i \Leftrightarrow <m(e_1), .., m(e_{n_i})> \in S_i)$. Let us explain this condition through our on-going example about control-flow complexity. Suppose that we have chosen m as a measure of control-flow complexity. The first part of the Rep-

resentation Condition says that, for any two program segments e_1, e_2, we have $<e_1,e_2> \in$ *more_complex_than* if and only if $m(e_1) > m(e_2)$.

Suppose we use the cyclomatic number as a measure of the control flow complexity of software code. Since the cyclomatic number of a program unit can be computed as the number of decision points in the software unit plus one (where an n-way decision point contributes as (n-1) two-way decision points), using the cyclomatic number as a software complexity scale implies that we rate as more complex a program unit *C10* with a single `case` statement with 10 exits than a program unit *W8* with 8 nested `while` loops. It is unclear if a wide consensus can be found on this ordering. If we believe that program unit *W8* is more complex than program unit *C10* we cannot use the cyclomatic number to measure control flow complexity.

This problem may not be due to a poor choice of a measure for a software attribute. Fenton [2] argues that no ordinal control-flow complexity scales exist because this would first imply finding a strict weak order (Definition 8) among the control-flow graphs according to their complexity and then finding a measure. The same situation is likely to occur for a number of software engineering attributes. Since we cannot find a strict weak order among the entities, we cannot define measures. We now provide the notion of strict weak order [5].

Definition 8. Strict Weak Order. A strict weak order is a pair $<E,R>$, where E is a set of entities and $R \subseteq E \times E$ is a binary relation that is

- Asymmetric: $\forall\ e_1,e_2 \in E\ R(e_1,e_2) \Rightarrow \neg R(e_2,e_1)$
- Negatively transitive: $\forall\ e_1,e_2,e_3 \in E\ \neg R(e_1,e_2) \wedge \neg R(e_2,e_3) \Rightarrow \neg R(e_1,e_3)$ or, equivalently, $\forall\ e_1,e_2,e_3 \in E\ R(e_1,e_2) \Rightarrow R(e_1,e_3) \vee R(e_2,e_3)$.

Strict weak orders can also be characterized through Theorem 1 [5], which requires the following notion of indifference relation.

Definition 9. Indifference Relation. Given a pair $<E,R>$, where E is a set of entities and $R \subseteq E \times E$, the relation $I \subseteq E \times E$ such that $I_R(e_1,e_2) \Leftrightarrow (\neg R(e_1,e_2) \wedge \neg R(e_2,e_1))$ is said to be the indifference relation derived from R.

Theorem 1. Strict Weak Order. A pair $<E,R>$, where E is a set of entities and $R \subseteq E \times E$, is a strict weak order if and only if R is

1. Asymmetric: i.e., $\forall\ e_1,e_2 \in E\ R(e_1,e_2) \Rightarrow \neg R(e_2,e_1)$
2. Transitive: i.e., $\forall e_1,e_2,e_3 \in E\ R(e_1,e_2) \wedge R(e_2,e_3) \Rightarrow R(e_1,e_3)$
3. and the indifference relation I_R is an equivalence relation.

Theorem 1 (from [5]) shows that the entities in a Strict Weak Order can be organized in a totally ordered set of equivalence classes, each of which contains elements that cannot be ordered. Not every partial order is a strict weak order. For instance, given $E = \{e_1,e_2,e_3,e_4\}$ and $R = \{<e_1,e_2>,<e_1,e_4>,<e_2,e_4>,<e_3,e_4>\}$, relation R is asymmetric and transitive, but it does not generate a relation I_R that is an equivalence relation as the pairs $<e_1,e_3>$ and $<e_2,e_3>$ belong to I_R, but $<e_1,e_2>$ does not.

It can be shown theoretically [5] that an ordinal scale exists iff the Empirical Relational System is a strict weak order. Fenton [2] contends that it would be impossible to find a sufficiently wide consensus on any negatively transitive order among control-flow graphs, so no ordinal control-flow complexity scales can possibly be defined. With reference to the control-flow graphs in Figure 1, it is argued that program segment *y* may be plausibly ranked as more complex than program segment *x*, but other pairs, such as *x* and *z* or *y* and *z* look incomparable. So, it would be impossible to find a wide consensus on a strict weak among these control-flow graphs.

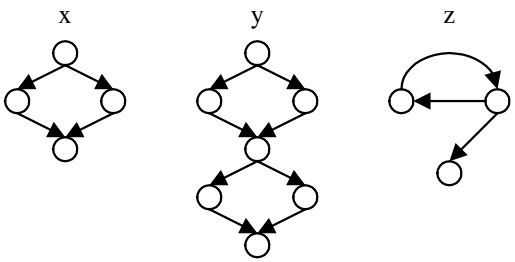


Fig. 1. Complexity ordering among control-flow graphs? (from [2])

Summarizing, we can provide an intuitive order for some of the pairs of entities, but not for all of them. In this situation, MT allows only for the existence of nominal measures. However, this implies a loss of information, since we have lost the piece of information about the ordering of the entities that we are actually able to order.

This is due to the nature of condition (1) of the Representation Condition, which states a property of the kind "if and only if," i.e., a double link is required between the Empirical Relational System and the Numerical Relational System. This double link may be too demanding for the current state of the art of software engineering measurement. It should be studied how this double link could be relaxed and what can be gained and lost by doing so, as we explain next.

3 Weak Measurement Theory

We propose a modified form of Measurement Theory, which we call Weak Measurement Theory (WMT for short). The difference between MT and WMT is that we do not require a double link between the Empirical Relational System and the Numerical Relational System in the Representation Condition, as follows.

Definition 10. Weak Representation Condition. A measure must satisfy the following two conditions

$$\forall i \in 1..n \; \forall <e_1, .., e_{n_i}> \in E^{n_i} \; (<e_1, .., e_{n_i}> \in R_i \Rightarrow <m(e_1), .., m(e_{n_i})> \in S_i) \quad (2)$$
$$\forall j \in 1..m \; \forall <e_1, e_2> \in E \times E \; (m(e_1 \; o_j \; e_2) = m(e_1) \bullet_j m(e_2))$$

This definition leads to the definition of weak scale.

Definition 11. Weak Scale. A weak scale is a triple <*ERS,NRS,m*>, where *ERS* is an Empirical Relational System, *NRS* is a Numerical Relational System, and *m* is a measure that satisfies the Weak Representation Condition.

In our complexity-related example, the Weak Representation Condition states that, for any two program segments e_1, e_2, whenever $<e_1,e_2> \in$ *more_complex_than* we have $m(e_1) > m(e_2)$ and $m(e_1 \oplus e_2) = m(e_1) + m(e_2)$. Since we have lost the implication from the Numerical Relational System to the Empirical Relational System, we also allow cases in which $<m(e_1), .., m(e_{n_i})> \in S_i$, but not $<e_1, .., e_{n_i}> \in R_i$, i.e., "false positives" are possible. In our complexity-related example, this means that we may have $m(e_1) > m(e_2)$, but not $<e_1,e_2> \in$ *more_complex_than*. We need to point out, however, that $<e_1,e_2> \notin$ *more_complex_than* does not imply $<e_2,e_1> \in$ *more_complex_than*. Instead, we do not have sufficient knowledge or we have not reached a sufficiently broad consensus on how to order e_1 and e_2 based on their complexity. As a proof of this statement, if we know that $<e_2,e_1> \in$ *more_complex_than*, then we must have $m(e_2) > m(e_1)$, due to condition (2) of the Weak Representation Condition and not $m(e_1) > m(e_2)$. Thus, having just the implication from the Empirical Relational System to the Numerical Relational System means that a measure must order correctly all those entities whose complexity is ordered in the Empirical Relational System. Whenever $<e_1,e_2> \notin$ *more_complex_than* and $<e_2,e_1> \notin$ *more_complex_than*, we are indifferent as for the relative values that the measures may associate with e_1 and e_2, i.e., we may indifferently have either have $m(e_1) > m(e_2)$, $m(e_1) = m(e_2)$, or $m(e_2) > m(e_1)$.

The presence of "false positives," also entails that, in the control-flow complexity example, when we have $m(e_1) > m(e_2)$, we cannot draw the conclusion that $<e_1,e_2> \in$ *more_complex_than* with certainty. Thus, there may be a loss of information when one uses the Weak Representation Condition. This potential loss should be compared against the benefits of using a less strict approach, as we now explain.

First, it should be borne in mind that, for a number of currently widely used measures, the Representation Condition simply does not hold. Two alternatives exist:

- discard all the measures that do not satisfy the original Representation Condition;
- study them and their properties with the Weak Representation Condition.

Second, MT is a relatively recent development and its usefulness should be assessed based on its applicability to real cases. It is a fact that, so far, MT has provided very little guidance in the definition of new software engineering measures and has had a mainly "destructive" function in finding out problems with the existing software engineering measures. We would like to add that the Weak form of MT has been suggested in the past as an alternative to MT [1], but, to the best of this author's knowledge, this avenue of research has not been pursued any further.

Third, in several cases, it is not possible to provide a strict weak order of the entities according to some attribute of interest, at least at this stage of the software measurement field. The examples of [2] are a clear proof of that. Multidimensional measurement has been suggested as an alternative in these cases. Instead of considering an attribute as a single one, it should be considered as composed of several different (sub)attributes, each of which can be measured independently. However, it may not be easy to identify the (sub)attributes that compose the attribute

easy to identify the (sub)attributes that compose the attribute of interest. Also, even if all the (sub)attributes were identified, a consensus on a strict weak order on the entities according to the overall attribute of interest cannot usually be obtained.

We propose instead that WMT be investigated so as to provide a less demanding mathematical theory as a foundation for measures. This implies providing new definitions in WMT for a number of concepts that exist in MT, starting from meaningful statement. The most natural way to provide a new definition of meaningful statement is to use the same notion as in Definition 6, except that the new definition involves replacing a weak scale with another weak scale.

Definition 12. Weak Meaningful Statement. A statement is called a weak meaningful statement if its truth value does not change if a weak scale is replaced by another weak scale. Formally, if $S(m)$ is a statement based on measure m and $S(m')$ is the same statement obtained by replacing m with m', we have $S(m) \Leftrightarrow S(m')$.

A weak meaningful statement is invariant across all weak scales. Based on the notion of weak meaningful statement, a theory of levels of measurement can be defined in WMT. Being more liberal than MT, Weak Measurement Theory imposes less strict constraints on the scales belonging to each category.

Weak Nominal Scales. The meaningful statements of this class of scales are of the form $m(e_1) = m(e_2)$, for at least one pair of entities e_1, e_2. If $m(e_1) = m(e_2)$ for a pair of entities e_1, e_2 and one scale m, we must have $m'(e_1) = m'(e_2)$ for all other scales m'. Whenever the set of values V of the measures has at least the same cardinality as the set of values of the entities E, we cannot impose that the entities e_1, e_2 be necessarily different. Otherwise, scales that are classified as nominal in MT would be excluded by the definition. In MT, when $|V| \geq |E|$, there are at least as many values as entities, so we can define a nominal measure just by assigning a different value to each entity. Thus, there may not exist two distinct entities e_1, e_2 such that $m(e_1) = m(e_2)$. However, when $|V| < |E|$, we are forced to provide the same value to at least two entities, in both MT and WMT. In both MT and WMT, values are labels for subsets of entities, with the following difference. In MT, the set of entities is partitioned in a set of equivalence classes, each of which receives a different label by every scale. In WMT, two different classes may receive the same label by some scale.

Weak Ordinal Scales. $\langle ERS, NRS, m \rangle$ is a weak ordinal scale if $m(e_1) > m(e_2)$ is a weak meaningful statement for at least one pair of entities e_1, e_2. It is not required that $m(e_1) > m(e_2)$ or $m(e_1) = m(e_2)$ be weak meaningful statements for all pairs of entities e_1, e_2. The values of these scales are labels for categories in which the entities are classified, with a *not necessarily total ordering* across the categories.

Weak Interval Scales. $\langle ERS, NRS, m \rangle$ is a weak interval scale if $(m(e_1)-m(e_2))/(m(e_3)-m(e_4))=k$ is a weak meaningful statement for at least one 4-tuple of entities e_1, e_2, e_3, e_4, i.e., k is a constant value across all scales. It is not required that this statement be meaningful for all 4-tuples of entities. Thus, we can quantify the difference between measurement values for some entities.

Weak Ratio Scales. $<ERS, NRS, m>$ is a weak ratio scale if $m(e_1)/m(e_2) = k$ is a weak meaningful statement for at least one pair of entities e_1, e_2, i.e., k is a constant value across all scales. Each entity is associated with a numerical value that provides a quantification for the ratio between some measurement values.

As admissible transformations across scales may not exist even in MT, we have defined the above levels of measurement based on the notion of weak meaningful statement. The notion of admissible transformation is less likely to be useful in WMT for this purpose, because the set of weak scales of WMT includes the set of scales of MT. Thus, the set of transformations that lead from one weak scale to another weak scale may not be easy to identify. For instance, the set of admissible transformations that lead from one weak ordinal scale to another includes all those transformations that preserve only the truth value of the meaningful statements, i.e., the ordering among those pairs of entities e_1, e_2 for which we must have $m(e_1)>m(e_2)$ for all measures m. Take two entities e_3, e_4 for which we need not have $m(e_3)>m(e_4)$ for all measures m, i.e., $m(e_3)>m(e_4)$ is not a weak meaningful statement. Given a measure m' for which $m'(e_3)>m'(e_4)$, there exists an admissible transformation of m' that leads to a measure m'' for which $m''(e_3)>m''(e_4)$ and another admissible transformation that leads to a measure m''' for which $m'''(e_4)>m'''(e_3)$.

At any rate, it is true that the admissible transformations used for the different levels of measurement of MT can be safely used to transform one weak scale into another weak scale in WMT. For instance, by applying any non-decreasing strictly monotonic transformation to a weak ordinal scale we still obtain a weak ordinal scale, since it certainly preserves the truth value of all meaningful statements.

4 Existence of Weak Scales for the Ordinal and Ratio Levels

Here, we first show how the theory of ordinal scales of MT can be extended to weak ordinal scales in WMT (Section 4.1). Then, we show how a theory of weak extensive measurement can be built in WMT (Section 4.2). In what follows, we need to use in the Empirical Relational Systems the concept of hierarchy, which we define next.

Definition 13. Hierarchy. A hierarchy is a pair $<E, R>$, where $R \subseteq E \times E$ is a binary relation on E such that it does not contain any cycle, i.e., any sequence of pairs $\{<e_1,e_2>, <e_2,e_3>, \ldots , <e_i,e_{i+1}>, \ldots , <e_n,e_{n+1}>\}$ of any length n with $\forall i \in 1..n\ R(e_i,e_{i+1})$ such that $e_1=e_{n+1}$.

For instance, hierarchy $<E,R>$ where $E = \{e_1, e_2, e_3, e_4, e_5, e_6\}$ and $R = \{<e_6,e_2>, <e_5,e_3>, <e_4,e_3>, <e_3,e_2>, <e_3,e_1>\}$ can be graphically represented as in Figure 2.

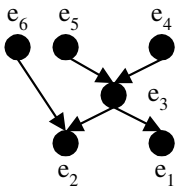


Fig. 2. A hierarchy

4.1 Weak Ordinal Scales

Based on operations research, Adams [1] proved Theorem 2, analogous to Theorem 1 for MT. (A different proof is in [4], with the proofs for all other theorems).

Theorem 2. **Necessary and Sufficient Condition for the Existence of a Weak Ordinal Scale.** Given the Empirical Relational System $ERS=<E,R>$, with $R \subseteq E \times E$ and the Numerical Relational System $NRS=<Real_+, >>$ ($Real_+$ is the set of positive real numbers), a weak ordinal scale $<ERS,NRS,m>$ exists iff R is a hierarchy.

When the implication from the Numerical Relational System to the Empirical Relational System is removed, the theorem shows that relation R of the Empirical Relational System may not be a strict weak order for a weak scale to exist. So, we have a scale type in WMT that is between the Nominal and the Ordinal ones of MT.

4.2 Weak Extensive Structure and Weak Ratio Scales

We prove a theorem on how we can even extend the results of Extensive Structures of MT to define Weak Extensive Structures in WMT and weak ratio scales.

Definition 14. Weak Extensive Structure. Let E be a set, $R \subseteq E \times E$ is a binary relation on E, and o a total function o: $E \times E \rightarrow E$. The triple $<E, R, o>$ is said to be a Weak Extensive Structure if and only if the following axioms hold.

A1 (Weak Associativity): $\forall\ e_1,e_2,e_3 \in E\ (Eq(e_1 o(e_2 o e_3),\ (e_1 o e_2) o e_3))$, where Eq is an equivalence relation defined as $Eq(e_1,e_2) \Leftrightarrow \neg R(e_1,e_2) \wedge \neg R(e_2,e_1)$.

Axiom A1 states that the order in which the entities in E are combined does not affect the ordering of the resulting elements. Thus, if $e_4=e_1 o(e_2 o e_3)$ and $e_5=(e_1 o e_2) o e_3$, it is neither $R(e_4,e_5)$ nor $R(e_5,e_4)$. The associativity of the combination operation is weak (like in MT) because it is not required that $e_1 o(e_2 o e_3)=(e_1 o e_2) o e_3$.

A2 (Hierarchy): $<E, R>$ is a hierarchy.

A3 (Monotonicity): $\forall\ e_1,e_2,e_3 \in E\ (R(e_1,e_2) \Rightarrow \neg R(e_2 o e_3, e_1 o e_3))$. The order between two entities e_1 and e_2 cannot be changed by combining e_1 and e_2 with any other entity e_3.

A4 (Archimedean): $\forall\ e_1,e_2,e_3,e_4 \in E\ (R(e_1,e_2) \Rightarrow \exists n \in \mathrm{N}\ \neg R(ne_2 o e_4, ne_1 o e_3))$ where ne is

recursively defined for any $e \in E$ as $1e=e$ and $\forall\, n > 1$ $ne=(n\text{-}1)e \circ e$. Axiom A4 states that, for any pair of entities e_3,e_4 and for any ordered pair of entities e_1,e_2, it is possible to find a number n such that the combination of e_1 n times with e_3 is not ordered with respect to the combination of e_2 n times with e_4.

Theorem 3 extends to WMT the theorem of the Extensive Structures for MT.

Theorem 3. Existence of an Additive Scale for a Weak Extensive Structure. Let E be a set, $R \subseteq E \times E$ a binary relation on E, and o a total function $o: E \times E \rightarrow E$. There is a function $m: E \rightarrow Real$, where $Real$ is the set of real values, such that $R(e_1,e_2) \Rightarrow m(e_1) > m(e_2) \wedge m(e_1 \circ e_2) = m(e_1) + m(e_2)$ if and only if $<E,R,o>$ is a Weak Extensive Structure.

Theorem 3 shows that there are several families of measures, one for each Strict Weak Order that can be defined on the transitive closure of R. This is different from MT, where there is only one such family of measures, and one measure m' can be obtained from any another measure m by multiplying m by a suitable coefficient. Each of the measures in WMT is an additive one. It is immediate to show that the scale built on top of an additive measure m is a weak ratio scale.

Theorem 4. Weak Additive Scales and Weak Ratio Scales. Let $<ERS, NRS, m>$ be a weak scale, with $ERS = <E, R, o>$ (where E is a set, $R \subseteq E \times E$, and o a total function $o: E \times E \rightarrow E$), $NRS = <Real, >, +>$, and m a function $m: E \rightarrow Real$, i.e., $R(e_1,e_2) \Rightarrow m(e_1) > m(e_2) \wedge m(e_1 \circ e_2) = m(e_1) + m(e_2)$. Then, $<ERS, NRS, m>$ is a weak ratio scale.

5 The Mean Value to Evaluate Central Tendency for Weak Scales

One of the most important characteristics of a set of measurements is its central tendency. Several indicators may be defined for measuring the central tendency of a set of measurements, e.g., the arithmetic mean and the median. In this section, we show when the arithmetic mean and the median can be used in WMT.

We would like to point out that, in MT, it is often said that the mean value of a set of ordinal measurements cannot be taken as an indicator of central tendency. The following example is typically used to show that the mean of an ordinal scale should not be used because it leads to meaningless statements. Suppose that m' is an ordinal measure, and that we have obtained two sets of failures $\{x_1, ..., x_n\}$ and $\{y_1, ..., y_p\}$ on two programs. Take statement $\frac{1}{n}\sum_{i \in 1..n} m'(x_i) > \frac{1}{p}\sum_{j \in 1..p} m'(y_j)$, which compares the mean values of the measures we have obtained on the two sets of failures. Conventional wisdom says that this statement is not meaningful since its truth value may depend on the specific ordinal scale chosen. In other words, it would not be true that

$$\frac{1}{n}\sum_{i \in 1..n} m'(x_i) > \frac{1}{p}\sum_{j \in 1..p} m'(y_j) \Leftrightarrow \frac{1}{n}\sum_{i \in 1..n} m''(x_i) > \frac{1}{p}\sum_{j \in 1..p} m''(y_j) \quad (3)$$

for every other ordinal measure m'' that we chose to use instead of m'. Examples can be easily built to show this lack of meaningfulness.

Surprisingly, this *general* statement is not true. In some cases, statement (3) is true independent of the ordinal measure chosen. To prove this property, which is against conventional wisdom, we proceed from more extreme less obvious examples and then we will build a general theory. Before doing so, we observe that the computation of the mean of an ordinal measure m with t values (e.g., a failure criticality measure that can assume $t = 5$ different values) can also be carried out as $\sum_{k\in 1..t} f_k m_k$, where f_k is the relative frequency with which value m_k is obtained. For instance, suppose that failures can be classified into five classes ($t = 5$) with the class whose index is $k = 1$ being the class of the least critical failures and the class corresponding to $k = 5$ being the class of the most critical failures. Note that k is just the index of the different classes, so it is not the measure chosen. Frequency f_k is the proportion of failures obtained by executing the program that have been assigned to class k, e.g., $f_1 = 0.2, f_2 = 0.25, f_3 = 0.35, f_4 = 0.1, f_5 = 0.1$ (the sum of the frequencies must be 1).

Thus, we can rewrite (3) as

$$\sum_{k\in 1..t} f_k m'_k > \sum_{k\in 1..t} g_k m'_k \Leftrightarrow \sum_{k\in 1..t} f_k m''_k > \sum_{k\in 1..t} g_k m''_k \quad (4)$$

where g_k is the frequency of class k failures obtained by executing a different program (or even by carrying out a different set of executions on the same program, with a different testing techniques, for instance).

Let us start with the most extreme example. Suppose we have obtained the following two frequency distributions: $f_1=0, f_2=0, f_3=0, f_4=0, f_5=1$ and $g_1=1, g_2= 0, g_3= 0, g_4=0, g_5=0$. Statement (4) is obviously true, since it becomes $m'_5>m'_1 \Leftrightarrow m''_5>m''_1$. Thus, *any* admissible (i.e., strictly monotonic) transformation of measure m' into a measure m'' will not change the truth value of the statement $m'_5 > m'_1$. Let us now move on to an even less extreme example, with the following two frequency distributions: $f_1 = 0.05, f_2 = 0.15, f_3 = 0.2, f_4 = 0.2, f_5 = 0.4$ and $g_1 = 0.1, g_2 = 0.15, g_3 = 0.25, g_4 = 0.3, g_5 = 0.2$. Statement (4) becomes

$$0.05m'_1+0.15m'_2+0.2m'_3+0.2m'_4+0.4m'_5 > 0.1m'_1 + 0.15m'_2 + 0.25m'_3+0.3m'_4+0.2m'_5$$

$$\Leftrightarrow$$

$$0.05m''_1+0.15m''_2+0.2m''_3+0.2m''_4+0.4m''_5>0.1m''_1+0.15m''_2+0.25m''_3+0.3m''_4+0.2m''_5$$

This statement is true, and the reason can be found in the following theorem.

Theorem 5. Let $\{w_1, ..., w_t\}$ and $\{m_1, ..., m_t\}$ be two sets of real numbers. Let the total ordering among the m_i's be known, and, let us suppose that $0 < m_1 < m_2 < ... < m_t$. We have $\sum_{k\in 1..t} w_k m_k > 0$ for every choice of $m_1, m_2, ..., m_t$ such that $0 < m_1 < m_2 < ... < m_t$ if and only if, for all $1\le q\le t$, the sum of the weights $\sum_{k\in q..t} w_k > 0$.

Theorem 5 can be used to compare two mean values by rewriting (4) as

$$\sum_{k\in 1..t} (f_k-g_k)\, m'_k > 0 \Leftrightarrow \sum_{k\in 1..t} (f_k-g_k)\, m''_k > 0$$

to apply Theorem 5 with $w_k = (f_k-g_k)$. Thus, it is not true in general that statement (3) is false in the general case even in MT. In WMT, we can prove a similar theorem for a special case of hierarchy (a forest), whose definition is based on the definition of tree.

Definition 15. Trees and Forests. A tree is a connected hierarchy in which (1) there exists only one node r (the root) such that there is no other entity a for which $<r,a> \in R$ and (2) for any node a_1 there is at most one other node a_2 such that $<a_1,a_2> \in R$. Two trees are disjoint if they do not share any nodes. A forest is the union of a set of disjoint trees. Two nodes c, d in the forest are said to be ordered if the pair $<c,d>$ belongs to the irreflexive transitive closure of R. Given a node d, the set of nodes c for which the pair $<c,d>$ belongs to the irreflexive transitive closure of R or is the pair $<d,d>$ is denoted by $Prec(d)$.

Theorem 6. Let $\{w_1, ..., w_t\}$ and $\{m_1, ..., m_t\}$ be two sets of real numbers. Let a forest ordering among the m_i's be known. We have $\sum_{k \in 1..t} w_k m_k > 0$ for every choice of non-negative $m_1, m_2, ..., m_t$ satisfying the given forest ordering if and only if, for all nodes d, we have $\sum_{k \in Prec(d)} w_k > 0$, i.e., where the sum of the w_k's is carried out over all k for which it is known from the forest ordering that $m_k > m_d$ plus the value w_d.

The result of Theorem 6 can be used to solve the problem in the general case of hierarchies. We first need to transform the hierarchy in a forest. The roots of the trees of the forest are the nodes of the hierarchy that have no successors. One tree will be built for each of these nodes. Then, a depth-first search is carried out from these roots. When a node a with more than one successor is found, replicate that node in each of the trees that are being built. The value of measure $m(a)$ is left unchanged, while the total weight $w(a)$ is distributed across all the new nodes that have been created. Thus, we introduce a set of variables representing the weights of the new nodes with the constraint that their sum must be equal to $w(a)$. When the visit of the hierarchy is completed, we have a forest and a set of linear equations that link the weights of the new nodes created. For instance, starting from the hierarchy in Figure 2, we obtain the forest represented in Figure 3.

So, the general problem is reduced to the following one: Given a finite set of linear equations (derived from the creation of the new nodes) and linear inequalities (derived by the requirements of Theorem 6) involving real valued variables, decide if the polytope determined by these constraints is empty. This problem has a Linear Programming algorithmic solution of relatively low computational cost, which is also the first part of the Symplex algorithm. In a time which is polynomial in the number of the variables we can decide if this set is empty. If not, the algorithm provides a solution point as well. Also, the entire process can be easily automated.

From the viewpoint of WMT, finding a solution to the above problem implies that the comparison between mean values is meaningful, i.e., its truth value does not change for any choice of the measure m' used.

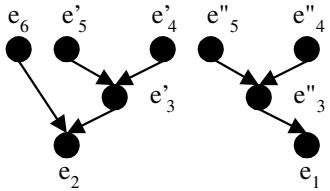


Fig. 3. Transformation of the hierarchy of Fig. 2 into a forest

6 Conclusions and Future Work

In this paper, we have presented a proposal for an approach to defining a theory of measurement with less strict requirements than MT. As such, WMT may be more easily applicable to the current state of the art of software measurement.

A more thorough investigation of the issues outlined in this paper is clearly called for, even though this paper may represent a consistent nucleus around which a full-fledged theory may be developed. Among the issues, we need to investigate further whether the notions of weak measurement levels is adequate or should be refined; how the properties of many important software engineering attributes, e.g., complexity, cohesion, coupling, may be modeled in the framework of WMT; what other gains and losses may result from the use of WMT instead of MT.

Acknowledgements. Special thanks to Stefano Serra-Capizzano for the useful discussions. This work was carried out with the partial support of MIUR and CNR.

References

1. E.W. Adams, "Elements of a Theory of Inexact Measurement" Philosophy of Science, Vol. 32, No. 3, pp. 205–228, July 1965.
2. N. Fenton, "Software Measurement: A Necessary Scientific Basis," IEEE Transactions on Software Engineering, Vol. 20, No. 3, pp. 199–206, March 1994.
3. D.H. Krantz, R. D. Luce, P. Suppes, A. Tversky, "Foundations of Measurement," Academic Press, New York, 1971.
4. S. Morasca, "A Weak Measurement-Theoretic Approach to Software Measurement," Università degli Studi dell'Insubria, Dip. di Scienze CC. FF. MM., Tech. Report. 2002.
5. F.S. Roberts, "Measurement Theory," Addison-Wesley, Reading, 1979.

An Information-Based View of Representational Coupling in Object-Oriented Systems

Pierre Kelsen*

Luxembourg University of Applied Sciences, Department of Applied Computer Science, L-1359 Luxembourg-Kirchberg

Abstract. In this paper we investigate a special type of coupling in object-oriented systems. When a method of a class C invokes a method of a class D, the method of C becomes dependent on the representational details of D: the more low-level the service provided by D is the higher the dependency of C on D. This dependency is known as *representational coupling*. Coupling in general, and representational coupling in particular, are important because they influence the extensibility of a system, that is, the ease with which software can be adapted to changing requirements: the higher the coupling the harder it is to make changes since any changes local to one module are likely to affect many other modules.

We propose a *qualitative measure* of representational coupling (as opposed to quantitative measures provided by metrics) that is based on partial orders over equivalence relations on the state space. We also introduce the notion of *intrinsic representational coupling* that expresses the amount of representational coupling that is inherent to the system. Finally, we show that despite its non-quantitative nature our measure can be useful in identifying candidate methods for refactoring. We demonstrate this by applying our measure to several examples in the literature, showing in each case how an implementation with non-minimal representational coupling can be transformed using a few simple refactorings into a solution with minimal representational coupling (equal to the intrinsic representational coupling).

Keywords. coupling, object-oriented, extensibility, refactoring, metrics

1 Introduction

Coupling is a well-known quality factor of software. It expresses how strongly individual software modules depend on each other. Coupling has a major impact on the extensibility of software, that is, on the ease with which software can be modified to adapt to changing requirements. The lower the coupling the easier it is to change individual modules since changes local to one module are likely

* Work supported by the Luxembourg Ministry of Culture, Higher Education and Research under grant IST/02/03

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 216–230, 2003.

to have a lower impact on other modules. There exists a similar relationship between coupling and fault-proneness, that is, the likelihood of detecting a fault in a class: the higher the coupling in a system, the higher the fault-proneness. The relationship between coupling and external quality attributes such as fault-proness has been demonstrated by defining metrics for various types of coupling and performing empirical studies on the relevance of these metrics (e.g., [BBM96, Bri97,LH93]).

In this paper we consider representational coupling, a fundamental type of coupling ubiquitous in object-oriented systems. Whenever an object calls a method of another object, this method call reveals something about the callee: in the worst case (resulting in high representational coupling) it discloses implementation level information, in the best case (low representational coupling) it provides high-level information that reveals little about the internal structure of the object. The term representational coupling was first used in [Cha93] and discussed in some detail in [Rich99].

For example if the invoked method is simply the accessor function for an attribute, we have a high degree of representational coupling since these accessor functions are close to the implementation level (and thus likely to change). On the other hand if the method that is invoked provides a much higher-level service, then implementation changes in that method‘s class are less likely to affect this service, thus resulting in a lower degree of representational coupling.

As we shall explain later in this paper existing coupling metrics (such as [ChiKem91,ChiKem94,LH93,Lee95,Bri97,Yac99,Ari02]) do not capture representational coupling. One reason for this is that they are based on counting different types of interactions. The resulting quantitative measures are too coarse-grained to evaluate the level of abstraction of method calls, something that is needed for evaluating representational coupling.

In this paper we develop a "qualitative measure" for representing representational coupling: rather than simply assigning an ordinal number to a given design, we base our measure on the refinement ordering over equivalence relations on the state space. We also develop the notion of intrinsic representational coupling, which is a measure for the minimum amount of representational coupling inherently contained in a system. We shall prove that no design can have representational coupling less than the intrinsic representational coupling.

Because of the qualitative nature of our measure it cannot be directly used as an indicator to make predictions on external quality factors (such as fault-proneness or extensibility). It can however be used to compare the representational coupling with the minimum representational coupling possible as given by the intrinsic representational coupling. In this manner our measure allows us to detect possible candidates for refactoring. We shall exemplify this application by considering three examples from the literature. For each of these examples we describe an ad-hoc solution that displays non-optimal representational coupling. By using a few simple refactoring techniques we are able to transform each of these implementations into a solution with representational coupling equal to the intrinsic representational coupling.

In Sect. 2 we introduce basic object-oriented terminology. In Sect. 3 we derive a mathematically precise definition of representational coupling. In Sect. 4 we define the notion of intrinsic representational coupling. In Sect. 5 we show by means of examples how we can achieve optimal coupling using simple refactorings. In the final section we present some conclusions of our work.

2 Object-Oriented Terminology

An object-oriented system is made up of *objects* which are instances of *classes*. Each class consists of *methods* and *attributes*. We consider only non-static methods. The allowed data types of attributes depend on the programming language. We shall simplify the following definitions by basing ourselves on Java. These definitions can easily be adapted to other languages (such as C++, Eiffel or Smalltalk).

Since we attempt in this paper to establish a sound mathematical basis for representational coupling, we need to exercise some care in defining the basic terms unambiguously. An *object* is really a triplet $(identifier \text{‹} type \text{‹} value)$ where the identifier is from a set of identifiers I and the value - which we also call the *state of the object* - is a tuple $[A_1 : v_1 \text{‹} \ggg A_k : v_k]$ where each v_i is the value of attribute named A_i. The value of an attribute is defined as follows: if the attribute is of a primitive type then v_i is the primitive value, if it is of a class type then the value is either null or an identifier in I, and finally, if it is of an array type, then its value is the tuple of values of the individual elements (recursive definition!). An *object set* consists of a set of objects such that different objects have distinct identifiers in I and each identifier of I occurs as identifier of an object in the set.

We refer to the pair (identifier set, object set) as the *system state*. The system starts out in an *initial state*. Any system state that can be reached from this initial state is called a *reachable state*. From here on whenever we talk about a state we imply that the state must be reachable.

3 Towards a Formal Definition of Representational Coupling

3.1 Related Work on Coupling

Much research effort has been spent on coupling in the field of software metrics. Several types of coupling in object-oriented systems have been investigated. In [Bri99a] three frameworks ([Eder94,HiMo95,Bri97]) for describing coupling are investigated and an attempt is made to unify those frameworks according to certain criteria. One of these criteria is the type of connection, that is, the mechanisms that may contribute to coupling. The following classification of connections is given (assume m is a method of class C and D denotes another class):

1. D is the type of an attribute of C
2. D is the type of a parameter or a return type of m
3. D is the type of a local variable of m
4. D is the type of a parameter of a method invoked by m
5. m references an attribute of D
6. m invokes a method of D
7. high-level dependency of C on D such as "uses"

Representational coupling corresponds to connection type 6. Connection types 1-4 involve class D as a type being used in C. Connection type 7 only measures coupling at a very high level. Of the two remaining types connection type 6 is in fact the only type of coupling that takes into account the methods of the server class. In [Bri99a] a list of metrics developed for each connection type is also given: from that list one can see that the largest number of metrics was indeed developed for connection type 6, thus underscoring its central role in coupling measurement.

Let us take a closer look at those metrics:

1. Coupling between Objects ([ChiKem91,ChiKem94]): counts the number of other classes to which a class is coupled.
2. Response for class ([ChiKem91,ChiKem94]): the number of methods that can be potentially executed in response to a message received by an object of that class.
3. Message passing coupling ([LH93]): number of send statements in a class.
4. Information-flow-based coupling([Lee95]): counts for a method of a class the number of polymorphically invoked methods, weighed by the number of parameters of the invoked method.
5. In [Bri97] a suite of measures is proposed that count various types of interactions (e.g., class-attribute, class-method, method-method,...).
6. More recently dynamic (runtime) metrics where developed ([Yac99,Ari02]): these are based on counting the number of messages sent (or received) by an object during runtime.

Without going into the details of the definitions of these measures we observe that they are all based on counting certain types of interactions. They are do not permit a deeper analysis of the interaction between a method and a specific class, something that is needed to "measure" the abstraction level of an interface used by a method.

3.2 Example: An Elevator Control System

En route to a more rigorous approach to representational coupling a concrete example will be of great help. The following example, taken from [Rich99] describes an elevator control system. The system contains two classes: ElevatorControl and Elevator. The class Elevator represents an elevator in a building. Each elevator has a direction and a position, yielding two methods direction() and position() in the Elevator class. The ElevatorControl class has a single responsibility, namely

handling requests. Thus, if somebody pushes a button on a certain floor this class has to dispatch an elevator to take care of that request. Accordingly ElevatorControl maintains a list of elevators and has a single method, handleRequest() that has as argument an object of class Request.

In the first implementation the *handleRequest()* function polls each elevator, asking for its direction and position. Based on this information the method computes the closest elevator and assigns the request to that elevator. Note that the information the ElevatorControl class asks from the Elevator class, namely direction and position, is rather low-level.

Here is the pseudo-code describing the first implementation:

Implementation 1 *Method ElevatorControl.handleRequest(Request r)*

```
1. float minDist = infty;// positive infinity
2. Elevator ec = null; // reference to closest elevator
2. for each elevator e do {
3.    compute d=distance(e,r) using e.direction() and e.position();
4.    if (d< minDist) {
5.       ec = e;
6.       minDist = d;
7.    }
8. }
9. if (ec!= null)
10. ec.addRequest(r);
```

In a second implementation the Elevator class offers a higher-level interface: it provides a single public function *computeDistance(Request r)* which computes the distance of this elevator to the request. In this implementation the handleRequest method simply asks each elevator to compute its distance from the request and then assigns, as before, the request to the closest elevator. All that changes in the implementation is that the pseudo-code on line 3 above is replaced by

```
d=e.computeDistance(r);
```

Intuitively in the second implementation the handleRequest method has lower representational coupling with respect to the Elevator class since it accesses a higher-level interface. This intuition is seconded by analyzing how changes to the Elevator class affect the system design. For instance suppose we want to take into account the fact that an elevator may be out of order. In this case we have to add some attribute to the ElevatorClass as well as a method outOfOrder(). In the first implementation we also have to adapt the implementation of the handleRequest function. In the second implementation it suffices to let the computeDistance return a very large number (e.g., positive infinity) to indicate that an elevator is out of order without affecting the ElevatorControl class.

Similarly if we want to consider service elevators that move more slowly or elevators that can only stop on certain floors, both the ElevatorControl and the

Elevator class need to be changed in Implementation 1 while the changes are restricted to the Elevator class in Implementation 2. (See [Rich99] for details.)

3.3 An Information-Based View

We take the previous example as the starting point towards a more formal definition. Recall that in Implementation 1 the method handleRequest() invokes the direction() and position() methods of the Elevator class while in the second implementation handleRequest() only uses the computeDistance() method of Elevator.

Let us compare the two implementations from an information-theoretic standpoint. The reason we can replace the calls to direction() and position() by a call to computeDistance() is that method handleRequest() does not actually need all the information conveyed by the first two low-level functions. Indeed we do not need to distinguish the two cases where an elevator has a different position and direction but the same distance to the request. This observation allows us to use the computeDistance() method since it conveys just enough information to the handleRequest() method.

Since a formalization based on the abstraction level of an interface seems difficult and since the informal notion of representational coupling appears to be related to the information exchanged between a method and a class, we put our measure of representational coupling on an information-theoretic base.

The concept of *message* is central to our analysis since it is the vehicle by which information between a method and a class is exchanged.

Definition 1 *A message is a 4-tuple* $(a, b, methodName, args)$ *where* a *and* b *are object identifiers,* $methodName$ *is the name of a method of object* b *and* $args$ *denotes the argument values for that method. Note that* $args$ *may itself be a tuple if the method takes several arguments. This notation means that the message is sent from object* a *to object* b *and invokes method* $b.methodName()$ *with arguments* $args$*.*

A method call may return a result. For the sake of uniformity, we shall assume that for every message that corresponds to a method call, there is a special *return message*. Thus, the return message for a message $(a, b, methodName, args)$ is a message $(b, a, return, result)$ where *return* is a reserved name and *result* is the result value of the function, or *void* if the function does not return a value.

For the following discussion we shall assume that a method m is executed on an object of class C. We shall analyze the information exchanged between method m and a second class D by examining the sequence of messages that enter or leave an object of D. For a given implementation of m this sequence depends on

- the system state s (see Sect. 2),
- the choice of an object c of class C (= object on which method m will be executed),

- the choice of an object d of class D (at which we observe the sequence of messages).

Here we assume that c and d are identifiers in the identifier set of system state s (see Sect. 2). We call the sequence of messages obtained for a given choice of s, c and d the *actual message sequence*. We shall call the function that assigns to each triplet (s, c, d) an actual message sequence the *message sequence function*. We also say that m *interacts with class* D *through message sequence function* q.

To illustrate these notions consider the elevator control system. In the first implementation the actual message sequence of an Elevator object consists of four messages: the position and direction query messages together with their return messages. In the second implementation the actual message sequence comprises only two messages: the computeDistance request and the return message with the result. Note that in both cases the actual message sequence depends on the system state and on the choice of the Elevator object since for different system states the Elevator object may have a different position and/or direction possibly resulting in different return messages (having different result fields). In neither case is the message sequence function a constant function.

To evaluate the amount of information exchanged between method m and object d, we could simply take into account the total size of all messages in the actual message sequence but such a measure would be highly dependent on the particular encoding chosen. Another problem with this approach is the necessity to deal with object references: how does one measure the information contained in an object reference?

We use a different approach based on examining the dependency of the actual message sequence on the internal state of the object d (at which we observe the sequence of messages).

Definition 2 *A message sequence function* q **induces an equivalence relation** *on* $S(D)$*, the set of possible states of an object of class* D *as follows: for* $s, s' \in S(D)$ *we define* $s \equiv_q s'$ *if and only if* $q(s_0, c, d) = q(s'_0, c, d)$ *(ie,* q *yields the same actual message sequence) for any choice of* c *and* d *(of types* C *and* D*) and for any two system states* s_0 *and* s'_0 *that differ only in the state of object* d *where they have values* s *and* s'*, respectively.*

Intuitively, two states of an object of D are equivalent if they cannot be distinguished by method m (since they produce the same actual message sequence regardless of the states of the other objects).

As noted earlier, in the elevator example the distance from the request can be computed from the position and direction of the elevator. We can express this in terms of the induced equivalence relations: the message sequence function for the first implementation induces an equivalence relation on the state space of the Elevator class that is a strict refinement of that induced by the message sequence function for the second implementation. We conclude that the elevator class reveals less information in the second implementation. (Recall that an equivalence relation is a *refinement* of another equivalence relation defined over

the same ground set if the equivalence classes of the former equivalence relation are subsets of the equivalence classes of the latter relation.)

Method m can be implemented in many different ways. In each case m interacts with class D through a possibly different message sequence function. We need to be able to compare these functions.

Definition 3 *Let q and q' be two message sequence functions. Message sequence function q is weaker than message sequence function q', written as $q \prec q'$ if q' induces an equivalence relation over $S(D)$ that is a strict refinement of the equivalence relation induced by q.*

We observe that this ordering on message sequence functions is not a partial order (since it is not anti-symmetric) but the refinement ordering on the corresponding equivalence relations is indeed a partial order.

We can apply this definition to representational coupling as follows: consider two implementations m and m' with "the same behavior" in a class C. (Two methods have the same behavior if from any possible initial state they produce the same final state.) Then we say that m has lower representational coupling than m' with respect to a class D if m interacts with D through a weaker message sequence function than m'.

4 Intrinsic Representational Coupling

Let $S_{m \lhd c}$ be the function that maps any system state to another system state based on the action of m invoked on object c. In a sense $S_{m \lhd c}$ describes the semantics of method m. That is, if we call $c \triangleright m()$ on initial system state s_0, then the system will be in state $S_{m \lhd c}(s_0)$ after m terminates. We call the function $S_{m \lhd c}$ the *state mapping* for method m. In this section we take the view that the same method may have different implementations yielding the same state mapping and thus having the same behavior. In this context it makes sense to speak of a method interacting with a class through different message sequence functions (corresponding to different implementations with the same behavior).

Suppose a method m interacts with class D through a certain message sequence function. How can we tell whether there is an implementation for m that yields an even weaker message sequence function? Or maybe even a weakest possible message sequence function. This would imply in a sense that m has some intrinsic degree of representational coupling.

In this section we give some answers to these questions. For a system state s and an object b of s, let $s - b$ denote the system state with the same object set as s, but without the state information for b (the states given for all other objects are the same as those in s).

Definition 4 *Method m induces an equivalence relation on the set $S(D)$ of possible states of d as follows: for $s \lhd s' \in S(D)$ we set $s \equiv_m s'$ if and only if for any two system states s_1 and s'_1 that differ only on d where they have states s and s', respectively, we have, for any choice of object c (of type C), $S_{m \lhd c}(s_1) - d = S_{m \lhd c}(s'_1) - d$.*

Loosely expressed this means that s and s' cannot affect the state outside d in different ways.

Theorem 1. *Suppose that m interacts with class D through message sequence function q. Then the equivalence relation induced by q (see definition 2) is a (not necessarily strict) refinement of the equivalence relation induced by m.*

Proof. Assume for a contradiction that this is not the case. Then there exist two states s and s' in $S(D)$ such that $s \equiv_q s'$ but $s \not\equiv_m s'$. In other words, there exist two system states s_1 and s'_1 that differ only on d where they have values s and s', respectively, and having the following two properties: (1) they produce the same actual message sequence, and (2) they yield a different state for an object other than d. But this is not possible: if the actual message sequence is the same then the states of objects other than d must remain the same also since the system looks exactly the same in both system states for any object other than d. QED

Lemma 1. *Suppose that m interacts with D through message sequence function q and that the equivalence relation induced by q on the state space of D is equal to the equivalence relation induced by m. Then m cannot interact with D through a message sequence function that is weaker than q.*

Proof. If m interacts with D through a weaker message sequence function, then the equivalence relation induced by that message sequence function is not a refinement of the equivalence relation induced by m thus contradicting the previous theorem. QED

If a message sequence function for an implementation satisfies the condition of the lemma, then we will say that the underlying implementation has *minimal representational coupling* with respect to class D equal to the *intrinsic representational coupling* of method m with respect to class D (implying that no other implementation can yield a weaker message sequence function).

5 Reducing Representational Coupling via Refactorings

The technique for reducing coupling is in principle quite simple: find a method in a class that interacts with another class through a non-optimal message sequence function; replace the method's implementation by a new implementation that has the same behavior but lower coupling.

More precisely we view the transformation as a 3-step process:

1. Find a method m of a class C that interacts with a class D using a non-optimal message sequence function q. For this it suffices to find two states $s1$ and $s2$ – which we shall call *witness states* – of an object d of D that are equivalent under the equivalence relation induced by m but not equivalent under the equivalence relation induced by q. In other words the two states yield different message sequences for d but they cannot affect the states of other objects differently. We may view such a pair of witness states as an indication that the coupling can be improved.

2. The second step consists in transforming the current implementation into one with lower coupling. The actual transformation will be done via simple refactorings. Which refactorings apply will depend on the case at hand but it will require manual inspection of the code.
3. The third step is optional: it consists in proving that the representational coupling of the new implementation is optimal. It assumes that the solution obtained through the transformation process has indeed optimal representational coupling equal to the intrinsic representational coupling.

We shall demonstrate this 3-step process by applying it to three examples from the literature. We use the following refactorings (in step 2):

- *ExtractMethod*: applied when a fragment of code can be grouped together; consists in turning the fragment into a method whose name explains the purpose of the method.
- *DecomposeConditional*: applied when there is a complicated conditional (if-then-else) statement; consists in extracting methods from the condition, then part, and else parts. Note: this refactoring may be viewed as a special case of the ExtractMethod refactoring where the code fragment is part of a conditional instruction.
- *MoveMethod*: applied when a method is using more features of another class than the class on which it is defined; consists in creating a new method with a similar body in the class it uses most and either turning the old method into a simple delegation, or removing it altogether.

Before we examine the examples, let us consider the relationship of the transformation process with existing work on refactoring. Most tools available for refactoring automate the process of safely applying a refactoring step; an example of such a tool is the Smalltalk Refactoring Browser ([RBJ97]). The problem of detecting candidates for refactoring is more difficult. The usual approach is based on manual inspection of the code to detect "bad smells" ([Fow99]). Only recently has a tool been developed that directly assists the developer in finding refactoring candidates; it is based on detecting invariants ([KEGN01]). Our method provides another way of finding candidates of refactoring that is based on a mathematically precise condition (see Lemma 1). In this sense we consider our approach to be complementary to existing approaches for finding refactoring candidates. Note however that we do not know at this point whether our condition for refactoring can be tested in an automated fashion.

5.1 Example 1: The Elevator Control System

We claim that in the first solution (Implementation 1) the handleRequest() method has non-optimal representational coupling with the Elevator class. To prove this we look for a pair of witness states for an elevator object: if a request is for a floor r other than the first or the last floor, then elevators at position $r-1$ going up and at position $r+1$ going down will be at the same distance

from the request. The position() and direction() queries will return different results but the states of other objects will not be affected differently. Thus the representational coupling is indeed non-optimal.

To obtain a solution with lower coupling, we first apply the ExtractMethod refactoring to the line of pseudo-code that computes the distance. The resulting computeDistance() method only uses features of the Elevator class. We can therefore apply a second refactoring step, the MoveMethod refactoring, to move this method to the Elevator class and delete the original method. By applying these two simple refactoring steps in sequence we arrive at the improved solution (given in Sect. 3.2).

We claim that in this second implementation the handleRequest() method has optimal representational coupling with respect to the Elevator class. We first show that the equivalence relation induced by handleRequest() has as equivalence classes the sets of states that yield the same distance to the request. This can be proven as follows: if two states yield the same distance, then they are treated the same by the state mapping. Conversely, two states with different distances $d_1 < d_2$ from the request can easily be distinguished: simply set the other states to a value that yields minimal distance d_1. Setting the state of the first elevator such that the distance is d_1 will result in it being closest (assume that in case of a tie the first elevator is chosen) and setting it to a state with distance d_2 will result in another elevator being closest.

Finally we observe that the message sequence function in the second implementation (yielding an actual message sequence consisting of the single computeDistance() query and its return value) induces exactly the same equivalence relation as the handleRequest() method. Thus we can apply Lemma 1, proving that the second implementation has indeed optimal representational coupling with respect to the Elevator class.

5.2 Example 2: The Heating System

This example is taken from [Rich99,Riel96] (in slightly adapted form). In a building a number of rooms are controlled by a central heating system. Each room has a current temperature, a desired temperature and an indication of whether it is occupied. The heating system will poll the rooms at regular time intervals and open the valve for the room thus providing it with heat if it is occupied and the current temperature is below the desired level.

In the first implementation there are three classes HeatControl, Room and Valve. The HeatControl class has a single method checkRooms() that performs the previously described polling process. The class Room has three methods *temperature()*, *desiredTemperature()* and *occupied()* returning the current temperature, the desired temperature and an occupancy flag. Finally the Valve class has a single method *command(c)* with an argument c that is either CLOSE or OPEN.

The checkRooms method works as follows: for each room it determines whether the room is occupied and whether the temperature is below the desired temperature by invoking the corresponding methods of the Room class. If

this condition is satisfied then HeatControl sends a open message to the valve for this room, otherwise it sends a close message to the valve.

Here is the pseudo-code for the checkRooms() method in the first implementation:

Implementation 2 *Method HeatControl.checkRooms()*

```
1. for each room r do {
2.     Valve v = r.getValve();
3.     if (r.temperature()<r.desiredTemperature() and r.occupied())
4.         v.command(OPEN);//send heat
5.     else
6.         v.command(CLOSE);//cut off heat
7. }
```

Let C denote the condition of the if-statement. Consider two states of a Room object that have different values for the individual queries in C but yield the same value for C. Two such states form a pair of witness states, showing that the checkRooms() method has non-optimal representational coupling with respect to the Room class.

How do we transform the first solution into one with lower representational coupling? We apply the DecomposeConditional refactoring to extract the condition in the if-then-else statement into a method *needsHeat*(). This method uses only features of the Room class and is therefore moved into that class using the MoveMethod Refactoring. This leads us to the second implementation in which the function checkRooms() asks each room whether it needs heat:

Implementation 3 *Method HeatControl.checkRooms()*

```
1. for each room r do {
2.     Valve v = r.getValve();
3.     if (r.needsHeat())
4.         v.command(OPEN);
5.     else
6.         v.command(CLOSE);
7. }
```

We note that the second implementation is more flexible than the first one: we can for instance change the rules under which a room is heated (e.g., heat only during certain time periods) by modifying only the Room class; in the first implementation both the Room class and the HeatControl class need to be modified.

We can apply Lemma 1 to show that the second implementation has minimal representational coupling w.r.t the Room class: indeed the equivalence relation induced by *checkRooms* on the state space of the Room class is the same as that induced by the message sequence function.

5.3 Example 3: A Stock Trading System

Both of the previous examples are control-oriented systems. This last example taken from [Rich99] is of a different kind. It represents a stock trading system. The system contains four classes of interest to us: ServiceRepresentative, TradingSystem, Order and OrderRegistry. For this discussion we assume that the ServiceRepresentative can cancel an order by invoking the cancelOrder() method of TradingSystem with the order number as argument. The Order class contains a method getStatus() that returns the current status of the order: open or not open (the latter choice applying when the order has been executed, it has expired or has been previously canceled). It also contains a cancel() method. The trading system can only cancel the order if it is open. In the first implementation the TradingSystem.cancelOrder()method first checks the status of the order; if it is open, it issues a cancel() message to the proper order.

Here is this implementation for the cancelOrder() method (we assume that OPEN is a constant of the Order class and TradingSystem.getOrderRegistry() returns a unique object of class OrderRegistry):

Implementation 4 *Method TradingSystem.cancelOrder(orderNumber)*
1. Order o = getOrderRegistry().getOrder(orderNumber);
2. if (o.getStatus()==Order.OPEN)
3. o.cancel();

Consider two states of an order object: open and not open. These two states yield different message sequences for the order object but yet they cannot affect the states of objects other than itself! It follows that the cancelOrder() method has non-optimal representational coupling w.r.t. the Order class.

To arrive at a solution with lower coupling, we first apply DecomposeConditional to extract the condition of the if-then-else statement into a separate method *canBeCanceled*(). That method uses only features of the Order class and is therefore moved to that class. Here is the resulting code:

Implementation 5 *Method TradingSystem.cancelOrder(orderNumber)*
1. Order o = getOrderRegistry.getOrder(orderNumber);
2. if (o.canBeCanceled())
3. o.cancel();

This second implementation is more flexible than the first one: indeed we can change the rule under which an order can be canceled. In Implementation 2 this only requires the implementation of canBeCanceled() to be modified in the Order class while in Implementation 1 the TradingSystem class is also affected since the method *TradingSystem::cancelOrder*() has to modified.

One thing has not changed though: the open and non-open states are still witnesses to the non-optimality of the implementation. Thus we need to look further for an optimal solution.

To achieve minimal representational coupling, we observe that line 2 and line 3 both use only features of the order object and can thus be moved using the ExtractMethod and MoveMethod refactorings to the Order class.

In other words lines 2 and 3 get replaced by a single line

o.cancel();

The cancel() method of the Order class is changed accordingly. The resulting implementation has obviously minimal representational coupling since the actual message sequence is independent of the state of the Order object.

6 Conclusions

In this paper we have proposed a mathematically precise definition of representational coupling based on the information exchanged between a method and a class via messages. We have also defined the notion of minimal representational coupling which expresses the minimal amount of representational coupling inherent in an implementation. Finally we explain, using several examples, how these notions can be used to identify candidate methods for refactoring and how they can guide us in the refactoring process.

There are quite a few open questions that need to be examined:

- Can we apply the transformation process described in the previous section to real-life examples consisting of several hundred or even thousands of classes? For this it would be helpful to automate the process (as it is done for software metrics). Because of the non-quantitative nature of our measure it is not clear whether this is possible.
- So far we have only considered the representational coupling with a single class. Can our approach be generalized to analyzing the representational coupling with several classes at once? Since a method of a class interacts typically with several other classes to perform its function, this is of course highly relevant if we want to apply our process to large software systems.
- What is the exact relationship with refactoring? Can a method with non-optimal coupling always be refactored to one with minimal coupling?

References

[Ari02] Erik Arisholm, Dynamic Coupling Measures for Object-Oriented Software, Proc. of the Eighth International Symposium on Software Metrics (Metrics'02), Ottawa, Canada, pp. 33–42.

[BBM96] V.R. Basili, L.C. Briand, W.L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicators, Transactions on Software Engineering 22(10), pp. 751–761, 1996.

[Bol95] T. Bollinger, "What Can Happen When Metrics Make the Call," Transactions IEEE Software, vol. 12, no. 1, Jan 1995.

[Bri97] L.C. Briand, P. Devanbu, and W. Melo, An investigation into coupling measures for C++, Proc. 19th Int'l Conf. Software Eng., ICSE'97, Boston, pp. 412–421, May 1997. .

[Bri99a] L.C. Briand, J.W. Daly, J.K. Wüst, A Unified Framework for Coupling Measurement in Object-Oriented Systems, IEEE Transactions on Software Engineering, vol. 25, No. 1, Jan/Feb 1999.

[Bri99b] L.C. Briand, J.K. Wüst, H. Lounis, Investigating quality factors in object-oriented designs: an industrial case study, Proceedings of the 21st international conference on Software engineering, pp. 345–354, 1999, Los Angeles, California, United States.

[Cha93] D. de Champeaux, D. Lea, P. Faure, Object-Oriented Systems Development, Reading, Mass.; Addison Wesley, 1993.

[ChiKem91] S.R. Chidamber and C.F. Kemerer, Towards a metrics suite for Object Oriented Design, Proc. Conf. Object-Oriented Programming: Systems, Languages and Applications, OOPSLA'91, Oct. 1991. Also published in SIGPLAN Notices, vol. 26, No. 11, pp. 197–211, 1991.

[ChiKem94] S.R. Chidamber and C.F. Kemerer, A metrics suite for Object Oriented Design, IEEE Trans. on Software Engineering, vol. 20, No. 6, pp. 476–493, 1994.

[Eder94] J. Eder, G. Kappel, and M. Schrefl, Coupling and Cohesion in Object-Oriented Systems, Technical Report, Univ. of Klagenfurt, 1994.

[Fow99] M. Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999.

[HiMo95] M. Hitz and B. Montazeri, Measuring Coupling and Cohesion in Object-Oriented Systems, Proc. Int'l Symp. Applied Corporate Computing, Monterrey, Mexico, Oct. 1995. A version of this paper (focusing on coupling only) has been published in *Object Currents*, vol. 1, No. 4, SIGS publications, 1996.

[Jon99] C. Jones, "Software Metrics: Good, Bad, and Missing," Computer, vol. 27, no. 9, Sept 1994.

[KEGN01] Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin, Automated support for program refactoring using invariants, ICSM 2001, Proceedings of the International Conference on Software Maintenance, Florence, Italy, November 6–10, 2001, pp. 736-743.

[Lee95] Y.S. Lee, B.-S. Liang, S.-F- Wu, and F.-J. Wang, Measuring the Coupling and Cohesion of an Object-Oriented Program based on information flow, Proc. Int'l Conf. Software Quality, Maribor, Slovenia, 1995.

[LH93] W. Li and S. Henry, Object-Oriented Metrics that Predict Maintanability, J. Systems and Software, vol. 23, no. 2, pp. 111–122, 1993.

[Rich99] C. Richter, Designing Flexible Object-Oriented Systems with UML, Macmillan Technical Publishing, 1999.

[Riel96] A. Riel, Object-Oriented Design Heuristics, Reading, Mass.: Addison-Wesley, 1996.

[RBJ97] D. Roberts, J. Brant and R. Johnson, A Refactoring tool for Smalltalk, Theory and Practice of Object Systems, 3(4), pp. 253–263, 1997.

[Yac99] Sherif M. Yacoub, Hany H. Ammar, and Tom Robinson, Dynamic Metrics for Object-Oriented Designs, Proc. of the sixth International Symposium on Software Metrics (Metrics'99), Boca Raton, Florida, USA, pp. 60–61.

A Temporal Approach to Specification and Verification of Pointer Data-Structures*

Marcin Kubica

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
fax: +48 22 5544400
kubica@mimuw.edu.pl

Abstract. We present a formalism for specification of pointer data-structures and programs operating on them, based on temporal specifications of dynamic algebras. It is an extension of first-order logic with temporal branching-time combinators. The use of this formalism is illustrated by examples. We also propose a Hoare-style calculus for verification of while-programs (operating on pointers) against specifications written in the proposed formalism, which is sound and complete in the sense of Cook.

1 Introduction

It is well known that pointer data-structures are widely used in implementation of efficient algorithms [4]. It is also well known that they are one of the main sources of programming errors. Therefore there is a need for formal verification of programs operating on pointers. On the other hand, pointers seem to be discriminated in the specification methods. There have been many attempts to incorporate pointers into the framework of Hoare logic (e.g. [2,3,11]). However, these cases are focused on verification of first-order specifications. They have limited applicability since first-order logic cannot express such basic properties as connectivity or existence of a cycle (cf. [17], Sect. 4). In case of [2], where the assertion language is a variant of second-order logic, specifications of common pointer data-structures are far from straightforward. We can also find specification methods based on monadic second-order logic [12,13] (granting decidability of many issues arising during verification), however their use concentrates on trees and lists.

We exploit the same analogy between pointer data-structures, graphs and transitive systems as in [10], where vertices are represented by sets of traces and assertions are second-order formulae on sets of traces. However, the formalism for specification of pointer data-structures proposed here is different. It is an adaptation of temporal specifications of dynamic algebras [5]—an extension of first-order logic with temporal branching-time combinators. In our approach we

* This research was supported by the Polish Scientific Research Committee (KBN) under grant 8T11C 01515.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 231–245, 2003.

interpret the notion of direct consequence in time as a pointer link. Hence, e.g., a possible history is an analog to a path of pointer links. We show on examples, that the presented formalism is concise and comprehensive. We also show a Hoare-style calculus for the verification of while-programs against specifications written in the presented formalism, which is sound and complete in the sense of Cook [14].

We assume that the reader is familiar with many-sorted algebras with partial functions (in short partial algebras) [7,18]. We allow overloading of function and variable names. Whenever it can cause ambiguities, the arity of a function symbol, or the sort of a variable is indicated. Whenever we use equality it is a strong equality, i.e. $t_1 = t_2$ is true iff both t_1 and t_2 are defined and their values are equal. The following notational convention is also used. Formula $t\downarrow$ is an abbreviation of $t = t$, and $t_1 \equiv t_2$ is an abbreviation of $t_1 = t_2 \vee (t_1 \neq t_1 \wedge t_2 \neq t_2)$. In other words, $t\downarrow$ means that the value of t is defined, and $\equiv$ is a weak equality, $t_1 \equiv t_2$ means that either both t_1 and t_2 are defined and their values are equal, or they are both undefined. If f is a (partial) function, then $f[a \rightarrow b]$ denotes such a (partial) function that $f[a \rightarrow b](a) = b$ and $\forall_{x \neq a} f(x) \equiv f[a \rightarrow b](x)$.

In Sect. 2 we model states and changes of pointer data-structures by partial many-sorted algebras. The formalism for specification of pointer data-structures and programs operating on them is described in Sect. 3. Section 4 contains definitions of syntax and semantics of while-programs. Hoare-style calculus for verification of while-programs operating on pointer data-structures is presented in Sect. 5. Section 6 contains final conclusions and remarks.

2 Modeling Data-Structures

In this section we describe how to model states and changes of states of pointer data-structures, using many-sorted partial algebras. We use Pascal, as an example programming language. Since we focus on common pointer data-structures, we consider only built-in types, records and pointer types. Our approach can be easily extended by arrays and enumeration types, however we omit them for simplicity of presentation.

The starting point of our considerations is the declaration of data-types. We do not allow complex declarations, i.e., data-type constructions cannot be nested. It allows us to identify all declared types by their names. We avoid type hiding. The interpretation of built-in types is not defined here, however we assume that it is known. We assume also that type `Boolean`, with its standard interpretation, is among built-in types. For the sake of simplicity we assume that the only operations allowed on built-in types are functions (without side-effects) and assignments. Let $\Sigma_B = \langle S_B, F_B \rangle$ be a fixed signature and B be a fixed partial Σ_B-algebra. We assume that sorts from B represent values of built-in types and functions from B model primitive operations on built-in types.

Definition 1. *Let D be a Pascal declaration of data-types of the form*

$$\texttt{Type } t_1 \texttt{ = } d_1\texttt{; } \ldots \; t_n \texttt{ = } d_n\texttt{;}$$

For $i = 1, \ldots, n$, if there exists such j that t_j is a type of pointers to t_i, then we denote t_j by $\uparrow t_i$ (if there are several such j, then we chose the smallest one). If D does not contain any type of pointers to t_i, then we treat $\uparrow t_i$ as a name of a new sort of pointers to t_i.

By $Sig(D) = \langle S, F \rangle$ we denote such a signature that:

$$S = S_B \cup \{t_1, \ldots, t_n, \uparrow t_1, \ldots, \uparrow t_n, \uparrow b_1, \ldots, \uparrow b_k\}, \quad F = F_B \cup \bigcup_{i=1,\ldots,n} f_i,$$

where $S_B = \{b_1, \ldots, b_k\}$ and f_i is defined in the following way:

- *if t_i is a type of pointers to a non-record type u, then*

$$f_i = \{\_\uparrow : t_i \to u, \texttt{Nil} :\to t_i\},$$

- *if t_i is a type of pointers to a record type u composed of the fields $p_1 : u_1, \ldots, p_l : u_l$, then*

$$f_i = \{\_\uparrow : t_i \to u, \texttt{Nil} :\to t_i, \_|p_1 : t_i \to \uparrow u_1, \ldots, \_|p_l : t_i \to \uparrow u_l\},$$

- *if t_i is a record type composed of the fields $p_1 : u_1, \ldots, p_l : u_l$, then*

$$f_i = \left\{ \begin{array}{c} \_\triangleright p_1 : t_i \to u_1, \ldots, \_\triangleright p_l : t_i \to u_l, \\ (p_1 = \_, \ldots, p_l = \_) : u_1 \times \ldots \times u_l \to t_i \end{array} \right\}.$$

We use partial $Sig(D)$-algebras to model states of data-structures declared by D. Addresses of variables (of type t) are represented by values of an appropriate pointer type ($\uparrow t$). If D does not contain declaration of such a type, then a new sort named $\uparrow t$ is introduced. Function symbols in $Sig(D)$ have the following intuitive meaning:

- `Nil` is a constant representing a null pointer,
- $\_\uparrow$ returns value pointed to by a pointer; this function is defined for addresses of allocated variables,
- $\_\triangleright p$ returns the value of the field p of a record,
- $(p_1 = \_, \ldots, p_l = \_)$ constructs record values from values of record fields,
- $\_|p$ returns the address of the field p of a given record.

Our intention is to model values of programming variables, by functions $\_\uparrow$. From here on, let D be the fixed type declarations. By *type* we mean any of the following sorts of $Sig(D)$: $b_1, \ldots, b_k, t_1, \ldots, t_n$.

Example 1. Let us consider the type declaration of singly-linked lists:

```
Type list = ↑elem;
     elem = record d: data; next: list end;
```

Let us denote these declarations by L. For the sake of simplicity we assume that `data` is a built-in type. $Sig(L)$ contains (among others):

- sorts: `list`, `elem`, `data`, ↑`list`, ↑`data`,

- function symbols:

$\_{\triangleright}\mathtt{d} : \mathtt{elem} \to \mathtt{data}$ $\quad \_\uparrow : \mathtt{list} \to \mathtt{elem}$ $\quad \mathtt{Nil} :\to \mathtt{list}$
$\_|\mathtt{d} : \mathtt{list} \to \uparrow\mathtt{data}$ $\quad \_\uparrow : \uparrow\mathtt{list} \to \mathtt{list}$ $\quad \mathtt{Nil} :\to \uparrow\mathtt{list}$
$\_{\triangleright}\mathtt{next} : \mathtt{elem} \to \mathtt{list}$ $\quad \_\uparrow : \uparrow\mathtt{data} \to \mathtt{data}$ $\quad \mathtt{Nil} :\to \uparrow\mathtt{data}$
$\_|\mathtt{next} : \mathtt{list} \to \uparrow\mathtt{list}$ $\quad (\mathtt{data} = \_\, \mathtt{list} = \_) : \mathtt{data} \times \mathtt{list} \to \mathtt{elem}$

We are not interested in all partial $Sig(D)$-algebras as models of states of data-structures. We can restrict our considerations to partial algebras satisfying some natural conditions regarding pointers and records.

Definition 2. *By $DStr(D)$ we denote a class of all partial $Sig(D)$-algebras A satisfying the following conditions:*

- *for each record type t composed of fields $p_1 : u_1, \ldots, p_l : u_l$, we have:*
  - *functions returning values of fields ($\_{\triangleright}p_i$) are total,*
  - *function $(p_1 = \_, \ldots, p_l = \_)$ constructing record values is total,*
  - *a set of record values is isomorphic with the Cartesian product of sets of values of its fields,*
- *for each sort t of pointers to u the function $\_\uparrow : t \to u$ is undefined for* `Nil`*,*
- *for each sort t of pointers to a record type r composed of fields $p_1 : u_1, \ldots, p_l : u_l$ we have:*
  - *function $\_|p_j : t \to \uparrow u_j$ (for $j = 1, \ldots, l$) is defined for all pointers different from* `Nil`*,*
  - *if $x : t$ is an address of a record, then function $\_\uparrow : \uparrow u_j \to u_j$ (for $j = 1, \ldots, l$) is defined for $x|p_j$ iff x is the address of an allocated record, i.e. $\forall_{x:t}(x\uparrow)\downarrow \Leftrightarrow (x|p_j\uparrow)\downarrow$,*
  - *the value of a field of the record pointed to by a pointer is the same as the value pointed to by the pointer to the field, i.e. $\forall_{x:t}(x\uparrow)\downarrow \Rightarrow x\uparrow{\triangleright}p_j = x|p_j\uparrow$,*
  - *different fields of the same record, of the same type, have different addresses, i.e. for $1 \le j, k \le l$, $j \ne k$, $u_j = u_k$ we have $\forall_{x:t} x|{\triangleright}p_j \ne x|{\triangleright}p_k$,*
  - *fields (of the same type) of different records of the same type have different addresses, i.e. for $1 \le j, k \le l$, $u_j = u_k$ we have $\forall_{x,y:t} x \ne y \Rightarrow x|p_j \ne y|p_k$,*
- *fields (of the same type) of records, whose addresses belong to different sorts, have different addresses, i.e. for two different sorts t, u of pointers to record types, respectively, r composed of fields $p_1 : r_1, \ldots, p_l : r_l$, and s composed of fields $q_1 : s_1, \ldots, q_m : s_m$, and $1 \le j \le l$, $1 \le k \le m$, $r_j = s_k$ we have $\forall_{x:t,y:u} x|p_j \ne y|q_k$.*

Note that, for a given D, all conditions stated above can be expressed as a first-order formula.

Partial algebras from $DStr(D)$ can be used to model states of data-structures. One should remember, that one partial algebra models a single state of a data-structure. Our intention is to model such properties as data-structure invariants and preconditions of programs operating on pointer data-structures, with subclasses of $DStr(D)$. We should note, however, that the partial algebras from

$DStr(D)$ cannot model changes of data-structures, e.g. postconditions of programs. To do it, we need to express modification of $\_\uparrow$ functions. For this purpose, we extend $Sig(D)$ so that it contains two copies of $\_\uparrow$ functions—one related to the state of a data-structure before modification, and one after modification. We adopt here the notational convention of decorating symbols, similar to one used, for example, in the specification method Z [15]. Undecorated functions $\_\uparrow$ refer to the 'current' state of the data-structure, i.e. 'after' modification. Functions $\grave{\_}\uparrow$ refer to the 'previous' state of the data-structure, i.e. before modification. Later on, we use also other, auxiliary, decorations $\_\uparrow^{\circledast}$, $\_\uparrow^{\odot}$, etc. Let us denote by $\mathcal{D}$ a fixed infinite set of all decorations. For sake of simplicity we sometimes consider undecorated $\_\uparrow$ functions, as decorated with an empty word $\square$.

Definition 3. *Let $\square$ be a signature and $\circledast$ be a decoration. By $\square^{\circledast}$ we denote a signature obtained from $\square$ by replacing all function symbols $\_\uparrow$ by $\_\uparrow^{\circledast}$. By convention $\square^{\square} = \square$.*

For a given $\square_1 \supseteq \square^{\circledast}$ we can treat $\circledast$ as a signature morphism $\circledast : \square \to \square_1$ renaming all $\_\uparrow$ functions to $\_\uparrow^{\circledast}$ and leaving all other function symbols and sort names unchanged. If $\square$ is a formula and t is a term then $\square^{\circledast} = \circledast(\square)$ and $t^{\circledast} = \circledast(t)$.

Let $Y \subseteq \mathcal{D} \cup \{\square\}$ be a set of decorations. By $\square^{Y}$ we denote $\square^{Y} = \bigcup_{\square \in Y} \square^{\square}$. Let $\square_1$ be a signature without $\backslash$ decoration. By $\square\,\square_1$ we denote the signature $\square_1 \cup \backslash\square_1$. If $\circledast \in Y$ and A is a partial $\square^{Y}$-algebra, then $A_{|\circledast}$ is a partial $\square$-algebra with $\_\uparrow$ functions equal to their $\_\uparrow^{\circledast}$ equivalents from A.

Definition 4. *Let $Y \subseteq \mathcal{D} \cup \{\square\}$, $\square \in Y$. We extend the definition of $DStr$ for $Sig(D)^{Y}$—by $DStr(Sig(D)^{Y})$ we denote the class of such partial $Sig(D)^{Y}$-algebras A, that for all decorations $\circledast \in Y$ we have $A_{|\circledast} \in DStr(D)$.*

Partial algebras from $DStr(\square\ Sig(D))$ can be used to model changes of data-structures declared by D. One should keep in mind, however, that one partial algebra models one possible change of a data-structure. Our intention is to model properties of programs operating on pointer data-structures with subclasses of $DStr(\square\ Sig(D))$.

Definition 5. *Let $Y \subseteq \mathcal{D} \cup \{\square\}$, $\square \in Y$ and $A \in DStr(Sig(D)^{Y})$. By $\overline{A}$ we denote the set of such partial algebras $A_1 \in DStr(Sig(D)^{Y})$ that A_1 differs from A only in interpretation of function symbols of the form $\_\uparrow^{\circledast}$ (for $\circledast \in Y$). By $|\overline{A}|$ we denote $|A|$.*

Intuitively, $\overline{A}$ determines the sorts and interpretation of function symbols not representing states of data-structures.

Let $A \in DStr(\square\ Sig(D))$. We can model such properties as postconditions of programs by subsets of $\overline{A}$. Data-structure invariants and possible changes of a data-structure are modeled by subsets of $\overline{A_{|\backslash}}$.

Dynamically allocated programming variables can be accessed only via pointers. Static and local programming variables can also be accessed by their names. We model allocation of these programming variables by variables valuations. For this purpose we adopt the following convention.

Definition 6. *Let* $Sig(D) = \langle S_{\mathbb{D}}, F_{\mathbb{D}} \rangle$, V *be a declaration of 'programming' variables of the following form* `Var` $x_1{:}t_1; \ldots x_n{:}t_n;$. *The set of variables induced by* V *is a many-sorted set of 'logical' variable names* $X = \langle X_s \rangle_{s \in S_{\mathbb{D}}}$, $X_s = \{\&x_i \mid \uparrow t_i = s\}$. *If the set of variables* X *is known from the context,* $\&x \in X_{\uparrow s}$, *then we write* x *as an abbreviation of* $\&x\uparrow$.

Intuitively, if x is a programming variable, then $\&x$ denotes the address of x. Note that the above convention allows us to describe the aliasing of programming variables, while we can write the name of a programming variable to reference its value. For the rest of this paper, let V be the fixed declaration of programming variables.

3 Temporal Specifications

Let us now consider the problem of specification of pointer data-structures and programs operating on them. It can be seen from Sect. 2, that this problem can be reduced (for given D, V, $\overline{A}$ and $v : X \to |\overline{A}|$) to specification of a subset of $\overline{A}$. Note that the first-order logic is too weak for this purpose, since it is known (cf. e.g. [17], Sect. 4) that we cannot express the notion of a path in it.

We often view pointer data-structures as graphs with labeled vertices and/or edges—records can be seen as vertices and pointers as edges. The analogy between such graphs, transitive systems and Kriepke models is obvious. Therefore it arises a natural question of usability of temporal logics for specification of pointer data-structures. The general idea of using temporal logics for the specification of abstract data-types is not new [5,8,16], however the idea of using it for the purpose of the specification of a pointer data-structure is novel.

One of the main notions appearing both in the specification of pointer data structures, and in transitive systems, is the notion of a path. In pointer data-structures a path is a sequence of records, which can be traversed by following pointer links. In transitive systems it represents a possible history of a computation. This analogy is also exploited in [10], where vertices are represented by sets of traces leading to them, and specifications are formulae on sets of traces.

In classical temporal logics, satisfaction of a temporal formula depends on the set of all possible paths (starting in a given state). In such a situation, it is not important, for example, whether two diverging paths cross again or not. On the contrary, in case of pointer data-structures, binary trees for example, it is crucial that for each record in a tree there is only one path leading to it from the root of the tree. Therefore classical temporal logics are of limited use for the specification of pointer data-structures. However, the formalism presented in [5] for the purpose of the specification of dynamic algebras does not have described disadvantages. Here we present its slightly modified version, adapted for our purposes.

Viewing a pointer data-structure as a graph, we often consider some of the pointers, abstracting from the rest of them. Usually we focus on pointers linking the records of a given type. In our approach, pointers are modeled by unary functions. We can describe edges representing selected pointers by a set of 'terms

with a hole'. Formally, the term with a hole is a term which can contain an additional, distinguished variable symbol '[]' called the hole. Such a term describes a set of edges in the following way: if we assign a source vertex to the hole and the value of the term is defined, then it is equal to the destination vertex. Since each record type is composed of a finite number of fields, there can be several, but limited, number of edges coming out of a vertex. Therefore, we can represent selected edges by a finite set of terms with a hole.

For the rest of this section, let $\square = \langle S_\square, F_\square \rangle$ be a signature and $X = \langle X_s \rangle_{s \in S_\square}$ be a many-sorted set of variables. (We assume $[] \notin X$.)

Definition 7. *Let $s \in S_\square$. Any finite set e of terms with a hole $[] : s$, $e \subseteq T_\square(X \cup \{[] : s\})_s$ is called a description (over $\square$) of edges between vertices of sort s. We denote the set of all such descriptions by $E_\square(X, s)$.*

Definition 8. *Let A be a partial $\square$-algebra, $s \in S_\square$, $v : X \to |A|$ be a variables valuation, and $e \in E_\square(X, s)$. We denote by $Path(A, s, v, e)$ such a set of (finite or infinite) sequences $p = \langle p_0, p_1, \ldots \rangle \in |A|_s^+ \cup |A|_s^\square$ (called paths) that:*

- *for all $0 \leq k \leq length(p) - 2$ there exists such $t \in e$ that $(v[[] \to p_k])^A(t) = p_{k+1}$, and*
- *if $p = \langle p_0, \ldots, p_k \rangle$, then, for all $t \in e$, $(v[[] \to p_k])^A(t)$ is undefined.*

If $p \in Path(A, s, v, e)$, $p = \langle p_0, \ldots, p_k, \ldots \rangle$, then by $B(p)$ we denote the first element of p, $B(p) = p_0$, and by $p_{|k}$ we denote the result of removing the first k elements from p, $p_{|k} = \langle p_k, \ldots \rangle$.

We distinguish two kinds of formulae: dynamic formulae and path formulae. The latter ones, however, are used only as parts of dynamic formulae.

Definition 9. *Let $s \in S_\square$. The sets of dynamic formulae $F_\square(X)$ and path formulae $P_\square(X, s)$ are defined by the mutual induction:*

- *if $s_1 \in S_\square$, $t_1, t_2 \in T_\square(X)_{s_1}$, then $t_1 =_s t_2 \in F_\square(X)$,*
- *if $\square_1, \square_2 \in F_\square(X)$, then $\square_1 \Rightarrow \square_2, \neg\square_1 \in F_\square(X)$,*
- *if $s_1 \in S_\square$, x is an identifier and $\square \in F_\square(X \cup \{x : s_1\})$, then $\forall_{x:s_1}\square, \exists_{x:s_1}\square \in F_\square(X)$,*
- *if $t \in T_\square(X)_s$, $\{e_1, \ldots, e_k\} \in E_\square(X, s)$ and $\square \in P_\square(X, s)$, then $\mathbf{A}_{t, e_1, \ldots, e_k}\square$, $\mathbf{E}_{t, e_1, \ldots, e_k}\square \in F_\square(X)$,*
- *if $\square_1, \square_2 \in P_\square(X, s)$, then $\square_1 \Rightarrow \square_2, \neg\square_1 \in P_\square(X, s)$,*
- *if $s_1 \in S_\square$, x is an identifier and $\square \in P_\square(X \cup \{x : s_1\}, s)$, then $\forall_{x:s_1}\square, \exists_{x:s_1}\square \in P_\square(X, s)$,*
- *if $\square \in F_\square(X \cup \{x : s\})$, then $[\square x \triangleright \square] \in P_\square(X, s)$,*
- *if $\square_1, \square_2 \in P_\square(X, s)$, then $\square_1 \mathcal{U} \square_2 \in P_\square(X, s)$,*

The dynamic formulae are interpreted for a given variables valuation, and the path formulae are interpreted for a given variables valuation and a path. Both dynamic and path formulae can be build using first-order combinators (with standard interpretation). Dynamic formula $\mathbf{A}_{t, e_1, \ldots, e_k}\square$ represents universal, and

$\mathbf{E}_{t\langle e_1\langle\rhd\rhd\rhd\langle e_k}\square$ represents existential quantification over paths starting in t and following edges from $e_1\langle\rhd\rhd\rhd\langle e_k$. In a path formula $[\square x\triangleright\square]$ variable x is assigned the first element of the path, over which it is interpreted. Operator $\mathcal{U}$ is a temporal operator 'until', interpreted along a given path.

De nition 10. *Let A be a partial $\square$-algebra $v : X \to |A|$ be a variables valuation, $s \in S_\square$, $\{e_1\langle\rhd\rhd\rhd\langle e_k\} \in E_\square\,(X\langle s)$, $\square \in Path(A\langle s\langle v\langle\{e_1\langle\rhd\rhd\rhd\langle e_k\})$, $\square \in F_\square\,(X)$ and $\square \in F_\square\,(X\langle s)$. The satisfaction of $\square$ in A under v (written $A \models_v \square$), and the satisfaction of $\square$ in A under v and $\square$ (written $A\langle\square \models_v \square$) are defined by mutual induction:*

- $\square$ $A \models_v t_1 = t_2$ *iff* $v^A(t_1)$ *and* $v^A(t_2)$ *are defined and* $v^A(t_1) = v^A(t_2)$,
- $\square$ $A \models_v \square_1 \Rightarrow \square_2$ *iff* $A \models_v \square_1$ *implies* $A \models_v \square_2$,
- $\square$ $A \models_v \neg\square$ *iff* $A \not\models_v \square$,
- $\square$ $A \models_v \forall_{x:s_1}(\square)$ $(A \models_v \exists_{x:s_1}(\square))$ *iff* $A \models_{v[x\to a]} \square$ *for all (some)* $a \in |A|_{s_1}$,
- $\square$ $A \models_v \mathbf{A}_{t\langle e_1\langle\rhd\rhd\rhd\langle e_k}\square$ $(A \models_v \mathbf{E}_{t\langle e_1\langle\rhd\rhd\rhd\langle e_k}\square)$ *iff* $v^A(t)$ *is defined, and for all (for some) paths* $\square \in Path(A\langle s\langle v\langle\{e_1\langle\rhd\rhd\rhd\langle e_k\})$ *(where s is a sort of t) such that* $B(\square) = v^A(t)$ *we have* $A\langle\square \models_v \square$,
- $\square$ $A\langle\square \models_v \square_1 \Rightarrow \square_2$ *iff* $A\langle\square \models_v \square_1$ *implies* $A\langle\square \models_v \square_2$,
- $\square$ $A\langle\square \models_v \neg\square$ *iff* $A\langle\square \not\models_v \square$,
- $\square$ $A\langle\square \models_v \forall_{x:s_1}\square$ $(A\langle\square \models_v \exists_{x:s_1}\square)$ *iff* $A\langle\square \models_{v[x\to a]} \square$ *for all (some)* $a \in |A|_{s_1}$,
- $\square$ $A\langle\square \models_v [\square x\triangleright\square]$ *iff* $A \models_{v[x\to B(\square)]} \square$,
- $\square$ $A\langle\square \models_v \square_1\mathcal{U}\square_2$ *iff there exists such* $j > 0$ *that* $\square_{|j}$ *is defined and* $A\langle\square_{|j} \models_v \square_2$, *and for all* $0 < i < j$ *we have* $A\langle\square_{|i} \models_v \square_1$.

We write $A \models \square$ iff $A \models_v \square$ for all $v : X \to |A|$.

We define boolean operators $\wedge$, $\vee$ and $\Leftrightarrow$ using $\Rightarrow$ and $\neg$ in a standard way. Similarly, we define temporal operators $\Box$ (everywhere on the path), $\Diamond$ (somewhere on the path) and $\mathcal{O}$ (next) using $\mathcal{U}$. We should stress out again that while using temporal analogies we do not consider possible histories, but we traverse a pointer data-structure along pointer links.

Let us consider the simple example of singly-linked lists.

Example 2. Let us recall type definitions from Example 1.

```
Type list = ↑elem;
     elem = record d:data; next:list end;
```

Heads of lists can be characterized by the following formula:

$$head(p) \Leftrightarrow (p\uparrow)\downarrow \wedge \forall_{q:\mathtt{list}}(q\uparrow\rhd\mathtt{next} \neq p)\ \triangleright$$

The invariant of a list data-structure can be described as follows:

$\square$ each head of a list is pointed to by some external pointer:

$$\forall_{l:\mathtt{list}} head(l) \Rightarrow \exists_{p:\uparrow\mathtt{list}}(p\uparrow = l \wedge \forall_{q:\mathtt{list}} q|\mathtt{next} \neq p)\ \langle$$

- each list terminates with a `Nil` pointer

$$\forall_{p:\texttt{list}} head(p) \Rightarrow \mathbf{A}_{p\texttt{‹}[]\Rightarrow\texttt{next}} \Diamond [\Box x \triangleright x = \texttt{Nil}]) \ \texttt{‹}$$

- each allocated record appears on some list

$$\forall_{p:\texttt{list}} (p\uparrow)\downarrow \Rightarrow \exists_{q:\texttt{list}} (head(q) \wedge \mathbf{E}_{q\texttt{‹}[]\Rightarrow\texttt{next}} \Diamond [\Box x \triangleright x = p])) \ \texttt{‹}$$

- two separate lists do not intersect:

$$\forall_{l_1\texttt{‹}l_2:\texttt{list}} (l_1 \neq l_2 \wedge head(l_1) \wedge head(l_2) \Rightarrow$$
$$\mathbf{A}_{l_1\texttt{‹}[]\Rightarrow\texttt{next}} \Box [\Box x \triangleright \mathbf{A}_{l_2\texttt{‹}[]\Rightarrow\texttt{next}} \Box [\Box y \triangleright x = y \Rightarrow x = \texttt{Nil}]]) \ \triangleright$$

Let us denote data-type invariant of lists, being the conjunction of the above conditions, by *list*. Let us consider a program replacing all values x appearing on a list p, with values y. We can specify this program by the following precondition:

$$list \wedge head(p) \wedge x\downarrow \wedge y\downarrow$$

and a postcondition:

$$\mathbf{E}_{\texttt{`}p\texttt{‹}[]\texttt{`}\uparrow\Rightarrow\texttt{next}} (\Box [\Box q \triangleright q\texttt{`}\uparrow\Rightarrow\texttt{d} = \texttt{`}x \Rightarrow q\uparrow\Rightarrow\texttt{d} = \texttt{`}y] \wedge$$
$$\forall_{q:\texttt{list}} ((q\texttt{`}\uparrow\Rightarrow\texttt{d} \neq \texttt{`}x \vee \Box [\Box r \triangleright r \neq q]) \Rightarrow q\uparrow \equiv q\texttt{`}\uparrow) \wedge$$
$$\forall_{r:\uparrow\texttt{data}} (\forall_{q:\texttt{list}}\ q|\texttt{d} \neq r) \Rightarrow r\uparrow \equiv r\texttt{`}\uparrow) \wedge \Box$$

where $\Box$ is a formula stating that variables of any types other, then `elem` and `data` do not change.

Example 3. Doubly-linked cyclic lists can have the following type-declarations:

```
Type list = ↑elem;
     elem = record d: data; prev, next: list end;
```

The following data-type invariant describes possible shapes of lists, and expresses the fact, that each list is pointed to by an external pointer:

$$\forall_{p:\texttt{list}} (p\uparrow)\downarrow \Rightarrow (\mathbf{A}_{p\Rightarrow\texttt{next}\texttt{‹}[]\Rightarrow\texttt{next}} \Diamond [\Box x \triangleright x = p] \wedge \mathbf{A}_{p\Rightarrow\texttt{prev}\texttt{‹}[]\Rightarrow\texttt{prev}} \Diamond [\Box x \triangleright x = p] \wedge$$
$$\exists_{q:\uparrow\texttt{list}} \forall_{r:\texttt{list}} (q \neq r|\texttt{next} \wedge q \neq r|\texttt{prev} \wedge$$
$$\mathbf{E}_{q\uparrow\texttt{‹}[]\Rightarrow\texttt{next}} \Diamond [\Box x \triangleright x = p]) \wedge$$
$$p\uparrow\Rightarrow\texttt{next}\uparrow\Rightarrow\texttt{prev} = p) \ \triangleright$$

Example 4. Binary directed acyclic graphs (in short: binary DAGs) can have the following type declarations:

```
Type bdag = ↑ node;
     node = record x: data; l, r: bdag end;
```

All possible shapes of binary DAGs can be described by the following formula:

$$bDAG \Leftrightarrow \forall_{p:\texttt{bdag}} ((p\uparrow)\downarrow \Rightarrow \mathbf{A}_{p\texttt{‹}[]\Rightarrow\texttt{l}\texttt{‹}[]\Rightarrow\texttt{r}} \Diamond [\Box x \triangleright x = \texttt{Nil}]) \ \triangleright$$

Binary trees can be seen as a sub-class of binary DAGs. Their possible shapes describes the following formula:

$$bDAG \ \wedge \ \forall_{p\texttt{‹}q\texttt{‹}r:\texttt{bdag}} (p\uparrow)\downarrow \wedge (q\uparrow\Rightarrow\texttt{r} = p \ \vee \ q\uparrow\Rightarrow\texttt{l} = p) \wedge$$
$$(r\uparrow\Rightarrow\texttt{r} = p \ \vee \ r\uparrow\Rightarrow\texttt{l} = p) \Rightarrow q = r \wedge \qquad \triangleright$$
$$\forall_{p:\texttt{bdag}} ((p\uparrow)\downarrow \wedge p\uparrow\Rightarrow\texttt{l} = p\uparrow\Rightarrow\texttt{p} \Rightarrow p\uparrow\Rightarrow\texttt{l} = \texttt{Nil})$$

4 Programming Language

In this section we present a language of while-programs operating on pointer data-structures, and its semantics. For this section, let X be the set of variables induced by the declaration V.

Definition 11. *We denote by $Prog_D(V)$ the set of Pascal blocks of instructions (correct with respect to D and V) which can be derived from the following grammar:*

while-program → `begin` *instructions* `end`
instructions → *instruction* | *instruction* ; *instructions*
instruction → □ | `begin` *instructions* `end` | *designator* := *expression* | `New(` *designator* `)` | `while` *expression* `do` *instruction* | `if` *expression* `then` *instruction* `else` *instruction*
designator → *identifier* | *designator* ▷ *identifier* | *expression* ↑

where expression (after replacing each programming variable x by $\&x\uparrow$) is a term of an appropriate type.

Designators are also called l-values. They represent addresses of variables. We define a translation of designators to terms of appropriate pointer sorts. The translation gives us the semantics of designators.

Definition 12. *Let o be a designator of a variable of type s. By $\&(o)$ we denote a term, of a sort of pointers to type s, defined inductively in the following way:*

- *if o is a name of a programming variable, then $\&(o) = \&o$,*
- *if o has a form $o_1 \triangleright p$, then $\&(o) = \&(o_1)|p$,*
- *if o has a form $o_1\uparrow$, then $\&(o) = o_1$.*

We can treat expressions appearing in a program as terms, and boolean expressions as logical formulae. However, we have to remember that if the value of a term is undefined, then the evaluation of an expression aborts the program execution. For this purpose we define a formula $OK(\square)$ which is true iff the evaluation of a boolean expression □ is defined.

Definition 13. *Let □ be a boolean expression. We denote by $OK(\square)$ a formula defined inductively in the following way:*

- *if □ = `true` or □ = `false`, then $OK(\square) = true$*
- *if □ = (□$_1$ `and` □$_2$), then $OK(\square) = OK(\square_1) \wedge (\neg\square_1 \vee OK(\square_2))$—analogously in case of other boolean operators,*
- *if □ = $(t_1 = t_2)$, then $OK(\square) = t_1\downarrow \wedge t_2\downarrow$,*
- *otherwise $OK(\square) = \square\downarrow$.*

We define an auxiliary function returning a state of the data-structure obtained by storing, in a given state, a given value under a given address.

Definition 14. *Let $A \in DStr(D)$, t_1 be a sort of pointers to t_2, $a \in |A|_{t_1}$, $b \in |A|_{t_2}$. We denote by $Store(A, a, b)$ a partial algebra $A_1 \in DStr(D)$ obtained from A by storing value b under address a.*

Note that $\_\uparrow^{A_1}_{t_1 \to t_2} = \_\uparrow^{A}_{t_1 \to t_2} [a \to b]$. Moreover, if t is a record type containing values of type t_2 and u is a sort of pointers to t, then $\uparrow\colon u \to t$ is modified accordingly. Also, if t_2 is a record type containing values of type t, then $\uparrow\colon\uparrow t \to t$ is modified accordingly, etc.

We define semantics of while-programs operationally, as a function returning, for a given starting state of a data-structure and variables valuation, a set of possible final states of the data-structure.

Definition 15. *Let $P \in Prog_D(V)$, $A \in DStr(D)$, $v : X \to |A|$ be a variables valuation. We denote semantics of P by $[\![P]\!]_v(A)$ and define it by induction on the structure of P:*

- $[\![\texttt{begin } P \texttt{ end}]\!]_v(A) = [\![P]\!]_v(A)$,
- $[\![Q; R]\!]_v(A) = \overrightarrow{[\![R]\!]_v}([\![Q]\!]_v(A))$,
- $[\![\square]\!]_v(A) = \{A\}$,
- $[\![o \texttt{ := } e]\!]_v(A) = \begin{cases} \{Store(A \langle v^A(\&(o)) \langle v^A(e))\} & \text{if } A \models_v o\downarrow \wedge\, e\downarrow \\ \emptyset & \text{otherwise} \end{cases}$,
- $[\![\texttt{New}(o)]\!]_v(A) = \left\{ A_1 \;\middle|\; \begin{array}{l} A \models_v o\downarrow \wedge \exists_{x \in |A|_{t_2} \langle y \in |A|_{t_3}} (\neg(x\uparrow^A)\downarrow \wedge \\ A_1 = Store(Store(A \langle v^A(\&(o)) \langle x) \langle x \langle y)) \end{array} \right\}$ *where t_1 is a sort of $\&(o)$, t_2 is a sort of o and t_2 is a sort of pointers to t_3,*
- $[\![\texttt{if } e \texttt{ then } Q \texttt{ else } R]\!]_v(A) = \begin{cases} [\![Q]\!]_v(A) & \text{if } A \models_v OK(e) \wedge e \\ [\![R]\!]_v(A) & \text{if } A \models_v OK(e) \wedge \neg e \\ \emptyset & \text{otherwise} \end{cases}$,
- $[\![\texttt{while } e \texttt{ do } P]\!]_v(A) = \left\{ B \;\middle|\; \begin{array}{l} \exists_{\langle A_0 \langle\langle\langle A_n \rangle \in \overline{A}^+} A_0 = A \;\wedge\; A_n = B \wedge \\ \bigwedge_{i=0 \langle\langle\langle n-1} (A_i \models_v (OK(e) \wedge e) \wedge \\ A_{i+1} \in [\![P]\!]_v(A_i)) \wedge A_n \models_v OK(e) \wedge \neg e \end{array} \right\}$ ▷

Note, that $[\![P]\!]_v(A) \subseteq \overline{A}$, i.e. the execution of a program can change only the contents of memory (represented by $\_\uparrow$ functions). All other functions and sorts remain unchanged.

5 Hoare Logic

In this section we deal with the problem of verification of while-programs operating on pointer data-structures against their specifications. A while-program specification can be expressed in the form of a precondition and a postcondition. We assume that these conditions are expressed in the formalism presented in Sect. 3. We present a version of Hoare logic [1,6,9] of while-programs, with dynamic formulae as assertions.

Usually, a postcondition is not only a condition on final states of the data-structure, but a relation between starting and final states. In the classical Hoare logic, this effect is achieved by the introduction of auxiliary (non-programming) variables. The precondition can state that a given programming variable is equal to an auxiliary one. Then the postcondition can refer to the previous value of the programming variable through the auxiliary one. However, in case of pointer data-structures, such auxiliary variables should store values of all dynamically

allocated variables. That would mean that assertions are expressed in second-order logic. Such an approach is presented in [2]. Note that these auxiliary variables are not quantified. Therefore they do not have to be variables. Instead of auxiliary variables, we allow the extension of the signature by introduction of function symbols $\_\uparrow$ with various decorations. We adopt the following notational convention:

- undecorated function symbols $\_\uparrow$ refer to the 'current' state of the data-structure,
- function symbols $\_\uparrow^{\setminus}$ can appear in postconditions and refer to the 'previous' state of the data-structure,
- function symbols $\_\uparrow$ decorated with other symbols are auxiliary.

For this section, let $Y \subseteq \mathcal{D} \setminus \{\setminus\} \cup \{\square\}$, $\square \in Y$, $\square = Sig(D)^Y$ and X be a set of variables induced by V.

Definition 16. *Each such triple* $\{\square\}P\{\square\}$ *that* $\square \in F_{\square}(X)$, $\square \in F_{\square\,\square}(X)$, $P \in Prog_D(V)$ *is called a dynamic Hoare formulae (over* $\square$ *and* V*). We denote the set of all such formulae (for a given* D, $\square$ *and* V*) by* $DH_{\square}(V)$.

In classical Hoare logic, satisfaction of a formula does not depend on (a single) valuation of variables. In our approach, logical variables represent addresses of programming variables and values of programming variables are represented by (undecorated and decorated) $\_\uparrow$ functions. Therefore we define satisfaction of dynamic Hoare formulae for the sets of partial algebras differing in interpretation of (undecorated and decorated) $\_\uparrow$ functions, and variables valuations.

Definition 17. *Let* $A \in DStr(\square\ \square)$, $v : X \to |A|$ *be a variables valuation,* $\square \in F_{\square}(X)$, $\square \in F_{\square\,\square}(X)$, $P \in Prog_D(V)$. *We denote by* $Post(\overline{A}, v, \square, P)$ *and* $Pre(\overline{A}, v, \square, P)$ *such sets of partial* $\square\ \square$*-algebras that:*

$$Post(\overline{A}, v, \square, P) = \{A_1 \in \overline{A} \mid A_1 \models_v {}^{\setminus}\square, A_1{}_{|\square} \in [\![P]\!]_v(A_1{}_{|\setminus})\},$$

$$Pre(\overline{A}, v, \square, P) = \{A_1 \in \overline{A} \mid \forall_{A_2 \in \overline{A}}(A_2 \text{ differs from } A_1 \text{ only in interpretation of function symbols } \_\uparrow, A_2{}_{|\square} \in [\![P]\!]_v(A_2{}_{|\setminus})) \Rightarrow A_2 \models_v \square\} \triangleright$$

Intuitively, *Post* defines a set of partial $\square\ \square$-algebras representing possible (terminating) executions of P starting in states satisfying $\square$. *Pre* defines a set of partial $\square\ \square$-algebras having such 'starting' states, that if P terminates, then $\square$ is true.

Definition 18. *We say that a dynamic Hoare formula* $\{\square\}P\{\square\} \in DH_{\square}(V)$ *is satisfied for* $\overline{A}$ *(*$A \in DStr(\square\ \square)$*) and variables valuation* $v : X \to |\overline{A}|$ *(written* $\overline{A} \models_v \{\square\}P\{\square\}$*), when for all* $A_1 \in Post(\overline{A}, v, \square, P)$, *we have* $A_1 \models_v \square$.

Proposition 1. *Let us assume that* $\overline{A} \models_v \{\square\}P\{\square\}$, $A_1 \in \overline{A}$, *then:*

- *if* $A_1 \models_v {}^{\setminus}\square$, *then* $A_1 \in Pre(\overline{A}, v, \square, P)$,
- *if there exists such* $\square \in F_{\square}(X)$ *that* $Pre(\overline{A}, v, \square, P) = \{A_1 \in \overline{A} \mid A_1 \models_v {}^{\setminus}\square\}$, *then for all* $A_1 \in \overline{A}$ *we have* $A_1 \models_v \square \Rightarrow \square$. □

Proposition 2. *If* $\square \in F_{\square}(X)$ *and* $\square_1 \square_2 \in F_{\square\,\square}(X)$ *such that* $Pre(\overline{A} v \square_1 P) = \{A_1 \in \overline{A} \mid A_1 \models_v \backslash\square\}$ *and* $Post(\overline{A} v \square P) = \{A_1 \in \overline{A} \mid A_1 \models_v \square_2\}$, *then* $\forall_{A_1 \in \overline{A}} A_1 \models_v \square_2 \Rightarrow \square_1$. □

Let $\circledast$ be a decoration. By $NChng(\circledast)$ we will denote a first-order formula stating that functions $\_\uparrow$ and $\_\uparrow^{\circledast}$ are identical (for all pointer sorts).

Let $p_1 \triangleright\triangleright\triangleright p_k s_1 \triangleright\triangleright\triangleright s_k \in S_{\square}$, p_i be a sort of pointers to s_i, $o_1 \in T_{\square}(X)_{p_1} \triangleright\triangleright\triangleright$, $o_k \in T_{\square}(X)_{p_k}$, $t_1 \in T_{\square}(X)_{s_1} \triangleright\triangleright\triangleright t_k \in T_{\square}(X)_{s_k}$ be terms without decorations. By $Stored(o_1 t_1 \triangleright\triangleright\triangleright o_k t_k)$ we will denote a first-order formula describing the results of storing the value of t_1 at o_1, t_2 at o_2, ..., t_k at o_k, i.e.: for all $A \in DStr(\square\,\square)$, $v : X \to |A|$ we have $A \models_v Stored(o_1 t_1 \triangleright\triangleright\triangleright o_k t_k)$ iff

$$A_{|\square} = Store(\triangleright\triangleright\triangleright(Store(A_{|\backslash} v^A(o_1) v^A(t_1)) \triangleright\triangleright\triangleright) v^A(o_k) v^A(t_k)) \triangleright$$

Obviously, such a formula always exists, but its form depends on declarations of record types. Note that if, for example, s is a record type, then $Stored(o t)$ has to express appropriate changes of record fields (and if some of these fields are also records, then changes of their fields, etc.). Therefore we do not give here a general form of *Stored*.

Now we present a deduction system for a dynamic Hoare logic.

Definition 19. *Let* $\square \subseteq F_{\square\,\square}(X)$ *be such a set of formulae (called assumptions), that for each signature morphism* $\square : \square\,\square \to \square\,\square$ *permuting decorations, if* $\square \in \square$, *then* $\square(\square) \in \square$. *Let* $\{\square\}P\{\square\} \in DH_{\square}(V)$. *We denote by* $\square \vdash \{\square\}P\{\square\}$ *the fact, that basing on assumptions from* $\square$ *and using the proof system described below, we can prove* $\{\square\}P\{\square\}$. *We will write* $\vdash \{\square\}P\{\square\}$ *instead of* $\emptyset \vdash \{\square\}P\{\square\}$.

□ *Axioms:*

$$\vdash \{\square\}\square\{\backslash\square \wedge NChng(\backslash)\} \quad (1)$$

$$\vdash \{\square\} o := e \{\backslash\square \wedge (\backslash o)\downarrow \wedge (\backslash e)\downarrow \wedge Stored(\backslash\&(o) \backslash e)\} \quad (2)$$

$$\vdash \{\square\}\mathtt{New}(o)\{\backslash\square \wedge (\backslash o)\downarrow \wedge \exists_{x:t_1 y:t_2}(\neg(x\backslash\uparrow)\downarrow \wedge Stored(\&(o) x x y))\} \quad (3)$$

where t_1 *is a sort of* o *and it is a sort of pointers to* t_2,

□ *Rules:*

$$\frac{\square \Rightarrow \square_1 \in \square \square_1 \vdash \{\square_1\}P\{\square_1\} (\square_1 \wedge \backslash\square) \Rightarrow \square \in \square}{\square \vdash \{\square\}P\{\square\}} \quad (4)$$

where $\square_1 \subseteq \square$,

$$\frac{\square \vdash \{\square \wedge NChng(\circledast)\}P\{\square\}}{\square \vdash \{\square\}P\{\square\}} \quad (5)$$

where $\circledast$ *is a decoration different from* $\backslash$, *and not appearing in* $\square$ *or* $\square$,

$$\frac{\square \vdash \{\square\}P\{\square\}}{\square \vdash \{\square\}\mathtt{begin}\ P\ \mathtt{end}\{\square\}} \quad (6)$$

$$\frac{\square \vdash \{\square\}P\{\square\} \square \vdash \{\square(\square)\}Q\{\square(\square)\}}{\square \vdash \{\square\}P;Q\{\square\}} \quad (7)$$

where $\square : \square\ \square \to \square\ \square$ *is a signature morphism changing* $\backslash$ *decorations to* $\circledast$*, and* $\circledast$ *is a decoration different from* $\backslash$ *and not appearing in* $\square$*,* $\square$ *or* $\square$*,*

$$\frac{\square \vdash \{\square \wedge OK(\square) \wedge \square\} P \{\square\} \quad \square \vdash \{\square \wedge OK(\square) \wedge \neg\square\} Q \{\square\}}{\square \vdash \{\square\} \texttt{if } \square \texttt{ then } P \texttt{ else } Q \{\square\}} \quad (8)$$

$$\frac{\square \vdash \{\square \wedge OK(\square) \wedge \square\} P \{\square\}}{\square \vdash \{\square\} \texttt{while } \square \texttt{ do } P \{\square \wedge OK(\square) \wedge \neg\square\}} \quad (9)$$

We denote the above proof system by DH.

Intuitively, $\square$ contains all known facts concerning data-structures, which does not depend on the actual state. For example, we can instantiate $\square$ with the set of dynamic formulae satisfied by all partial algebras from a given $\overline{A}$ (for a given variables valuation v), i.e. $\square = Th_{\square\,\square}(\overline{A} \cup v) = \{\square \in F_{\square\,\square}(X) \mid \forall_{A_1 \in \overline{A}} A_1 \models_v \square\}$.

Theorem 1. *The proof system DH is sound, i.e. if* $\square \vdash \{\square\} P \{\square\}$*,* $A \in DStr(\square)$ *and* $v : X \to |A|$ *are such, that for all* $\square \in \square$ *and* $A_1 \in \overline{A}$ *we have* $A_1 \models_v \square$*, then* $\overline{A} \models_v \{\square\} P \{\square\}$

Proof of this theorem can be found in [14].

The proof system *DH* is also complete in the sense of Cook.

Definition 20. *Let* $A \in DStr(\square\ \square)$ *and* $v : X \to |A|$ *be variables valuation. We say that the set of dynamic formulae* $F_{\square\,\square}(X)$ *is expressive, with respect to* $\overline{A}$*,* v *and* $Prog_D(V)$*, if for all* $\square \in F_{\square\,\square}(X)$ *and* $P \in Prog_D(V)$ *there exists such* $\square \in F_{\square}(X)$ *that* $Pre(\overline{A} \cup v \cup \square \cup P) = \{A_1 \in \overline{A} \mid A_1 \models_v \square\}$.

Theorem 2. *The proof system DH is complete in the sense of Cook, i.e. for all* $A \in DStr(\square\ \square)$ *and* $v : X \to |A|$*, if* $F_{\square\,\square}(X)$ *is expressive, with respect to* $\overline{A}$*,* v *and* $Prog_D(V)$*, and* $\overline{A} \models_v \{\square\} P \{\square\}$*, then, for* $\square$ *being natural insertion of* $\square\ \square$ *into* $Sig(D)^{\mathcal{D}}$ *and such* $A_1 \in DStr(Sig(D)^{\mathcal{D}})$ *that* $A_{1\,|\square} = A$*, we have* $Th_{Sig(D)^{\mathcal{D}}}(\overline{A_1} \cup v) \vdash \{\square\} P \{\square\}$.

The proof of this theorem can be found in [14]. It follows the classical schema presented e.g. in [1], with all the necessary adaptations.

6 Conclusions

This work is a step forward in merging specification methods with efficient algorithms and data-structures. We have adopted temporal specifications of dynamic algebras [5] for the purpose of specification of pointer data-structures and programs operating on them. In the proposed formalism temporal combinators have been used to describe structure of pointer links. As it can be seen from examples, this idea allows compact and relatively readable specification of pointer data-structures. Its practical usefulness should be verified by thorough case-studies.

We have embedded the specification formalism within the framework of Hoare logic. Presented proof system satisfies standard properties of soundness and completeness in the sense of Cook. This gives us an elementary tool for verification of programs operating on pointer data-structures.

References

[1] K.R. Apt. Ten years of Hoare's logic: A survey — part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, Oct. 1981.

[2] H. Bickel and W. Struckmann. The Hoare logic of data types. Technical Report 95-04, Technische Universität Braunschweig, Deutschland, 1995.

[3] R. Cartwright and D. Oppen. The logic of aliasing. *Acta Inf.*, 15:365–384, 1981.

[4] T.H. Cormen, C.E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[5] G. Costa and G. Reggio. Specification of abstract dynamic-data types: A temporal approach. *Theoretical Comput. Sci.*, 173(2):513–554, 1997.

[6] O.-J. Dahl. *Verifiable Programming*. Prentice Hall, 1992.

[7] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specificaton 1*. Number 6 in EATCS MTCS. Springer-Verlag, 1985.

[8] Y. Feng and J. Liu. A temporal approach to algebraic specifications. In J. C. M. Baeten and J. W. Klop, editors, *CONCUR'90: Theories of Concurrency: Unification and Extension*, number 458 in LNCS, pages 216–229. Springer-Verlag, 1990.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–583, 1969.

[10] C.A.R. Hoare and H. Jifeng. A trace model for pointers and objects. In *ECOOP'99*, number 1628 in LNCS, pages 1–18. Springer-Verlag, 1999.

[11] T.M.V. Janssen and P. van Emde Boas. On the proper treatment of referencing, dereferencing and assignment. In G. Goos and J. Hartmanis, editors, *Automata, Languages and Programming*, number 52 in LNCS, pages 282–300. Springer-Verlag, 1977.

[12] N. Klarlund and M.I. Schwartzbach. Graph types. In *Proceedings of the 20th Symposium on Principles of Programming Languages*, pages 196–205. ACM, 1993.

[13] N. Klarlund and M.I. Schwartzbach. Graphs and decidable transductions based on edge constraints. In S. Tison, editor, *Trees in Algebra and Programming — CAAP'94. Proceedings*, number 787 in LNCS, pages 187–201. Springer-Verlag, 1994.

[14] M. Kubica. Temporal-style specifications of pointer data-structures. Technical Report TR 02–03 (268), Institute of Informatics, Warsaw University, 2002.

[15] J.M. Spivey. *The Z notation, A Reference Manual*. Second edition. Prentice-Hall, 1992.

[16] A. Szałas. Towards the temporal approach to abstract data types. *Fundamenta Informaticae*, XI:49–63, 1988.

[17] W. Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, volume 3, chapter 7. Springer-Verlag, 1997.

[18] M. Wirsing. Algebraic specifications. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 13, pages 676–788. Elsevier, 1990.

A Program Logic for Handling JAVA CARD's Transaction Mechanism

Bernhard Beckert[1] and Wojciech Mostowski[2]

[1] Institute for Logic, Complexity, and Deduction Systems
University of Karlsruhe, Germany
beckert@ira.uka.de
[2] Chalmers University of Technology, Göteborg, Sweden
Computing Science Department
woj@cs.chalmers.se

Abstract. In this paper we extend a program logic for verifying JAVA CARD applications by introducing a "throughout" operator that allows us to prove "strong" invariants. Strong invariants can be used to ensure "rip out" properties of JAVA CARD programs (properties that are to be maintained in case of unexpected termination of the program). Along with introducing the "throughout" operator, we show how to handle the JAVA CARD transaction mechanism (and, thus, conditional assignments) in our logic. We present sequent calculus rules for the extended logic.

1 Introduction

Overview. The work presented in this paper is part of the KeY project [1,9]. One of the main goals of KeY is to provide deductive verification for a real world programming language. Our choice is the JAVA CARD language [6] (a subset of JAVA) for programming smart cards. This choice is motivated by the following reasons. First of all JAVA CARD applications are subject to formal verification, because they are usually security critical (e.g., authentication) and difficult to update in case a fault is discovered. At the same time the JAVA CARD language is easier to handle than full JAVA (for example, there is no concurrency and no GUI). Also, JAVA CARD programs are smaller than normal JAVA programs and thus easier to verify. However, there is one particular aspect of JAVA CARD that does not exist in JAVA and which requires the verification mechanism to be extended with additional rules and concepts: the persistency of the objects stored on a smart card in combination with JAVA CARD's transaction mechanism (ensuring atomicity of bigger pieces of a program) and the possibility of a card "rip out" (unexpected termination of a JAVA CARD program by taking the smart card out of the reader/terminal). Since we want to have support for the full JAVA CARD language in the KeY system we have to handle this aspect.

To ensure that a JAVA CARD program is "rip-out safe" we need to be able to specify "strong" invariants—invariants that must hold throughout the whole execution of a JAVA CARD program (except when a transaction is in progress). The KeY system's deduction component uses a program logic, which is a version of

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 246–260, 2003.

Dynamic Logic modified to handle JavaCard programs (JavaCard DL) [2,3]. An extension to pure Dynamic Logic to include trace modalities "throughout" and "at least once" is presented in [4]. Here we extend that work and introduce the "throughout" operator to JavaCard DL (we do not introduce "at least once" since it is not necessary for handling "rip out" properties). Then we add techniques necessary to deal with the JavaCard transaction mechanism (specifically conditional assignments inside the transactions). We present the sequent calculus rules for our extensions. So far we have not implemented the new rules in the KeY system's interactive prover (the implementation for the unextended JavaCard DL is fully functional). But considering the extensibility and open architecture of the KeY prover it is not a difficult task.

Related Work. As said above, the work presented here is based on [4], which extends pure Dynamic Logic with trace modalities "throughout" and "at least once". There exist a number of attempts to extend OCL with temporal constructs, see [5] for an overview. In [16] temporal constructs are introduced to the Java Modelling Language (JML), but they refer to sequences of method invocations and not to sequences of intermediate program states.

Structure of the Paper. The rest of this paper is organised as follows. Section 2 gives some more details on the background and motivation of our work and some insights into the JavaCard transaction mechanism. Section 3 contains a brief introduction to JavaCard Dynamic Logic. Section 4 introduces the "throughout" operator in detail and presents sequent calculus rules to handle the new operator and the transaction mechanism. Section 5 shows some of the rules in action by giving simple proof examples and finally Sect. 6 summarises the paper.

2 Background

The KeY Project. The main goal of the KeY project [1,9] is to enhance a commercial CASE tool with functionality for formal specification and deductive verification and, thus, to integrate formal methods into real-world software development processes. Accordingly, the design principles for the software verification component of the KeY system are: (1) The specification language should be usable by people who do not have years of training in formal methods. The Object Constraint Language (OCL), which is incorporated into the current version of the Unified Modelling Language (UML), is the specification language of our choice. (2) The programs that are verified should be written in a "real" object-oriented programming language. We decided to use JavaCard (we have already stated our reasons for this decision in the introduction).

For verifying JavaCard programs, the already mentioned JavaCard Dynamic Logic has been developed within the KeY project (Sect. 3 contains a detailed description of this logic). The KeY system translates OCL specifications into JavaCard DL formulas, whose validity can then be proved with the KeY system's deduction component.

Motivation. The main motivation for this work resulted from an analysis of a JAVA CARD case study [11]. In short, the case study involves a JAVA CARD applet that is used for user authentication in a Linux system (instead of a password mechanism). After analysing the application and testing it, the following observation was made: the JAVA CARD applet in question is not "rip-out safe". That is, it is possible to destroy the applet's functionality by removing (ripping out) the JAVA CARD device from the card reader (terminal) during the authentication process. The applet's memory is corrupted and it is left in an undefined state, causing all subsequent authentication attempts to be unsuccessful (fortunately this error causes the applet to become useless but does not allow unauthorised access, which would have been worse).

It became clear that, to avoid such errors, one has to be able to specify (and if possible verify) the property that a certain invariant is maintained at all times during the applet's execution, such that it holds in particular in case of an abrupt termination. Standard UML/OCL invariants do not suffice for this purpose, because their semantics is that if they hold before a method is executed then they hold after the execution of a method. Normally it is not required for an invariant to hold in the intermediate states of a method's execution. To solve this problem, we introduce "strong" invariants, which allow to specify properties about all intermediate states of a program.

For example, the following "strong" invariant (expressed in pseudo OCL) says that we do not allow partially initialised `PersonalData` objects at any point in our program. In case the program is abruptly terminated we should end up with either a fully initialised object or an uninitialised (empty) one:

```
context PersonalData throughout:
  not self.empty implies
    self.firstName <> null and self.lastName <> null and self.age > 0
```

Since the case study was explored in the context of the KeY project, we extended the existing JAVA CARD DL with a new modality to handle strong invariants.

The JAVA CARD Transaction Mechanism. Here we describe the aspects of transaction handling in JAVA CARD relevant to this paper. A full description of the transaction mechanism can be found in [6,13,14,15].

The memory model of JAVA CARD differs slightly from JAVA's model. In smart cards there are two kinds of writable memory: persistent memory (EEPROM), which holds its contents between card sessions, and transient memory (RAM), whose contents disappear when power loss occurs, i.e., when the card is removed from the card reader. Thus every memory element in JAVA CARD (variable or object field) is either persistent or transient. The JAVA CARD language specification gives the following rules (this is a slightly simplified view of what is really happening): All objects (including the reference to the currently running applet, `this`, and arrays) are created in persistent memory. Thus, in JAVA CARD all assignments like "`o.attr = 2;`", "`this.a = 3;`", and "`arr[i] = 4;`" have a permanent character; that is, the assigned values will be kept after the card

loses power. A programmer can create an array with transient elements, but currently there is no possibility to make objects (fields) other than array elements transient. All local variables are transient.

The distinction between persistent and transient objects is very important since these two types of objects are treated in a different way by JavaCard's transaction mechanism. The following are the JavaCard system calls for transactions with their description:

`JCSystem.beginTransaction()` begins an atomic transaction. From this point on, all assignments to fields of objects are executed conditionally, while assignments to transient variables or array elements are executed unconditionally (immediately).

`JCSystem.commitTransaction()` commits the transaction. All conditional assignments are committed (in one atomic step).

`JCSystem.abortTransaction()` aborts the transaction. All the conditional assignments are rolled back to the state in which the transaction started. Assignments to transient variables and array elements remain unchanged (as if there had not been a transaction in progress).

As an example to illustrate how transactions work in practice, consider the fragment of a JavaCard program shown on the right. After the execution of this program, the value of `this.a` is still 100 (value before the transaction), while the value of `i` now is 100 (the value it was updated to during the transaction).

```
this.a = 100;
int i = 0;
JCSystem.beginTransaction();
  i = this.a;
  this.a = 200;
JCSystem.abortTransaction();
```

Transactions do not have to be nested properly with other program constructs, e.g., a transaction can be started within one method and committed within another method. However, transactions must be nested properly with each other (which is not relevant for the current version of JavaCard, where the nesting depth of transactions is restricted to 1).

The whole program piece inside the transaction is seen by the outside world as if it were executed in one atomic step (considering the persistent objects). By introducing strong invariants we want to ensure the consistency of the persistent memory of a JavaCard applet, thus strong invariants will not (and should not) be checked within a transaction—in case our program is terminated abruptly during a transaction, the persistent variables will be rolled back to the state before the transaction was started for which the strong invariant was established.

3 JavaCard Dynamic Logic

Dynamic Logic [7,8,10,12] can be seen as an extension of Hoare logic. It is a first-order modal logic with modalities $[p]$ and $\langle p \rangle$ for every program p (we allow p to be any sequence of JavaCard statements). In the semantics of these modalities a world w (called state in the DL framework) is accessible from the current world,

if the program p terminates in w when started in the current world. The formula $[p]\Box$ expresses that $\Box$ holds in *all* final states of p, and $\langle p\rangle\Box$ expresses that $\Box$ holds in *some* final state of p. In versions of DL with a non-deterministic programming language there can be several such final states (worlds). Here, since JAVA CARD programs are deterministic, there is exactly one such world (if p terminates) or there is no such world (if p does not terminate). The formula $\Box \rightarrow \langle p\rangle\Box$ is valid if, for every state s satisfying precondition $\Box$, a run of the program p starting in s terminates, and in the terminating state the post-condition $\Box$ holds. The formula $\Box \rightarrow [p]\Box$ expresses the same, except that termination of p is not required, i.e., $\Box$ must only hold *if* p terminates.

3.1 Syntax of JAVA CARD DL

As said above, a dynamic logic is constructed by extending some non-dynamic logic with modal operators of the form $\langle\cdot\rangle$ and $[\cdot]$. The non-dynamic base logic of our DL is a typed first-order predicate logic. We do not describe in detail what the types of our logic are (basically they are identical with the JAVA types) nor how exactly terms and formulas are built. The definitions can be found in [2]. Note that terms (which we often call "logical terms" in the following) are different from JAVA expressions—they never have side effects.

The programs in DL formulas are basically executable JAVA CARD code. However, we introduced an additional construct not available in plain JAVA CARD, whose purpose is the handling of method calls. Methods are invoked by syntactically replacing the call by the method's implementation. To treat the `return` statement in the right way, it is necessary (a) to record the object field or variable x that the result is to be assigned to, and (b) to mark the boundaries of the implementation *prog* when it is substituted for the method call. For that purpose, we allow statements of the form `method_call(`x`){`*prog*`}` to occur. This is a "harmless" extension because the additional construct is only used for proof purposes and never occurs in the verified JAVA CARD programs.

3.2 Semantics of JAVA CARD DL

The semantics of a program p is a state transition, i.e., it assigns to each state s the set of all states that can be reached by running p starting in s. Since JAVA CARD is deterministic, that set either contains exactly one state (if p terminates normally) or is empty (if p does not terminate or terminates abruptly).

For formulas $\Box$ that do not contain programs, the notion of $\Box$ being satisfied by a state is defined as usual in first-order logic. A formula $\langle p\rangle\Box$ is satisfied by a state s if the program p, when started in s, terminates normally in a state s' in which $\Box$ is satisfied. A formula is satisfied by a model M, if it is satisfied by one of the states of M. A formula is valid in a model M if it is satisfied by all states of M; and a formula is valid if it is valid in all models. Sequents are notated following the scheme $\Box_1 \text{‹} \rhd\rhd\rhd \text{‹} \Box_m \vdash \Box_1 \text{‹} \rhd\rhd\rhd \text{‹} \Box_n$ which has the same semantics as the formula $(\forall x_1)\cdots(\forall x_k)((\Box_1 \wedge \rhd\rhd\rhd \wedge \Box_m) \rightarrow (\Box_1 \vee \rhd\rhd\rhd \vee \Box_n))$, where $x_1 \text{‹} \rhd\rhd\rhd \text{‹} x_k$ are the free variables of the sequent.

3.3 State Updates

We allow *updates* of the form $\{x := t\}$ resp. $\{o{\triangleright}a := t\}$ to be attached to terms and formulas, where x is a program variable, o is a term denoting an object with attribute a, and t is a term. The intuitive meaning of an update is that the term or formula that it is attached to is to be evaluated after changing the state accordingly, i.e., $\{x := t\}\Box$ has the same semantics as $\langle$`x = t;`$\rangle\Box$.

3.4 Rules of the Sequent Calculus

Here we only present a small number of rules necessary to get proper intuition of how the JavaCard DL sequent calculus works.

Notation. The rules of our calculus operate on the first *active* statement p of a program $\Box p\Box$. The non-active prefix $\Box$ consists of an arbitrary sequence of opening braces "`{`", labels, beginnings "`try{`" of `try-catch-finally` blocks, and beginnings "`method_call(`$\triangleright\triangleright\triangleright$`){`" of method invocation blocks. The prefix is needed to keep track of the blocks that the (first) active statement is part of, such that the abruptly terminating statements `throw`, `return`, `break`, and `continue` can be handled appropriately. The postfix $\Box$ denotes the "rest" of the program, i.e., everything except the non-active prefix and the part of the program the rule operates on. For example, if a rule is applied to the Java block "`l:{try{ i=0; j=0; }finally{ k=0; }}`", operating on its first active statement "`i=0;`", then the non-active prefix $\Box$ is "`l:{try{`" and the "rest" $\Box$ is "`j=0; }finally{ k=0; }}`".

In the following rule schemata, $\mathcal{U}$ stands for an arbitrary update.

The Rule for `if`*.* As the first simple example, we present the rule for the `if` statement:

$$\frac{\Box \circ \mathcal{U}(b \stackrel{\triangleright}{=} \mathit{true}) \;\vdash\; \mathcal{U}\langle\Box p\Box\rangle\Box \qquad \Box \circ \mathcal{U}(b \stackrel{\triangleright}{=} \mathit{false}) \;\vdash\; \mathcal{U}\langle\Box q\Box\rangle\Box}{\Box \;\vdash\; \mathcal{U}\langle\Box \texttt{if}(b)\{p\}\,\texttt{else}\,\{q\}\,\Box\rangle\Box} \tag{R1}$$

The rule has two premisses, which correspond to the two cases of the `if` statement. The semantics of this rule is that, if the two premisses hold in a state, then the conclusion is true in that state. In particular, if the two premisses are valid, then the conclusion is valid. In practice rules are applied from bottom to top: from the old proof obligation new proof obligations are derived. As the `if` rule demonstrates, applying a rule from bottom to top corresponds to a symbolic execution of the program to be verified.

The Assignment Rule and Handling State Updates. The assignment rule

$$\frac{\Box \;\vdash\; \mathcal{U}\{\mathit{loc} := \mathit{expr}\}\langle\Box\,\Box\rangle\Box}{\Box \;\vdash\; \mathcal{U}\langle\Box\,\texttt{loc = expr;}\,\Box\rangle\Box} \tag{R2}$$

adds the assignment to the list of updates $\mathcal{U}$. Of course, this does not solve the problem of computing the effect of an assignment, which is particularly

complicated in JAVA because of aliasing. This problem is postponed and solved by rules for simplifying updates.

The assignment rule can only be used if the expression *expr* is a logical term. Otherwise, other rules have to be applied first to evaluate *expr* (as that evaluation may have side effects). For example, these rules replace the formula $\langle$`x = ++i;`$\rangle\Box$ with $\langle$`i = i+1; x = i;`$\rangle\Box$.

4 Extension for Handling "Throughout" and Transactions

In some regard JAVA CARD DL (and other versions of DL) lacks expressivity—the semantics of a program is a relation between states; formulas can only describe the input/output behaviour of programs. JAVA CARD DL cannot be used to reason about program behaviour not manifested in the input/output relation. Therefore, it is inadequate for verifying strong invariants that must be valid throughout program execution.

Following [4], we overcome this deficiency and increase the expressivity of JAVA CARD DL by adding a new modality $[\![\cdot]\!]$ ("throughout"). In the extended logic, the semantics of a program is the sequence of all states its execution passes through when started in the current state (its *trace*). Using $[\![\cdot]\!]$, it is possible to specify properties of the intermediate states of terminating and non-terminating programs. And such properties (typically strong invariants and safety constraints) can be verified using the JAVA CARD DL calculus extended with additional sequent rules for $[\![\cdot]\!]$ presented in Sect. 4.1.

A "throughout" property (formula) has to be checked after every single field or variable assignment, i.e., the sequent rules for the throughout modality will have more premisses and branch more frequently. According to the JAVA CARD runtime environment specification [14], each single field or variable assignment is atomic. This matches exactly JAVA CARD DL's notion of a single update. Thus, a "throughout" property has to hold after every single JAVA CARD DL update. However, additional checks have to be suspended when a transaction is in progress. This will require marking the modality (resp. the program in the modality) with a tag saying that a transaction is in progress, so that different rules apply. Since transactions do not have to be nested properly with other program constructs, enclosing a transaction in a block with a separate set of rules for that kind of block (like the `method_call` blocks) is not possible.

In addition, we have to cover conditional assignments and assignment rollback (after `abortTransaction`) in the calculus. This not only affects the "throughout" modality, but the $\langle\cdot\rangle$ and $[\cdot]$ modalities as well, since rolling back an assignment affects the final program state.

In practice only formulas of the form $\Box \rightarrow [\![p]\!]\Box$ will be considered. If transient arrays are involved in $\Box$ (explicitly or implicitly), one also has to prove $\Box \rightarrow \langle$`initAllTransientArrays();`$\rangle\Box$, i.e., that after a card rip-out the reinitialisation of transient arrays preserves the invariant.

4.1 Additional Sequent Calculus Rules for the $[\![\cdot]\!]$ Modality

Below, we present the assignment and the `while` rules for the $[\![\cdot]\!]$ modality. Due to space restrictions, we cannot list all additional rules. However, the other loop rules are very similar to the `while` rule, and all other $[\![\cdot]\!]$ rules are essentially the same as for $[\cdot]$—except for the transaction rules which we present in the next subsection.

The Assignment Rule for $[\![\cdot]\!]$. An assignment `loc = expr;` is an atomic program, if *expr* is a logical term (and, in particular, is free of side effects and can be computed in a single step). By definition, its semantics is a trace consisting of the initial state s and the final state $s' = \{loc := val_s(expr)\}s$. Therefore, the meaning of $[\![$`loc = expr;`$]\!]\Box$ is that $\Box$ is true in both s and s', which is what the two premisses of the following assignment rule express:

$$\frac{\Box \vdash \mathcal{U}\Box \quad \Box \vdash \mathcal{U}\{loc := expr\}[\![\Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\Box \texttt{ loc = expr; } \Box]\!]\Box} \tag{R3}$$

The left premiss states that the formula $\Box$ has to hold in the state s before the assignment takes place. The right premiss says that $\Box$ has to hold in the state s' after the assignment—and in all states thereafter during the execution of the rest $\Box$ of the program.

It is easy to see that using this rule causes some extra branching of the proofs involving the $[\![\cdot]\!]$ modality. This branching is unavoidable due to the fact that the strong invariant has to be checked (evaluated) for each intermediate state of the program execution. However, many of those branches, which do not involve JAVA CARD programs any more, can be closed automatically.

The `while` *Rule for* $[\![\cdot]\!]$. Another essential programming construct, where the rule for the $[\![\cdot]\!]$ modality differs from the corresponding rule for the $[\cdot]$ modality, is the `while` loop. As in the case of the `while` rule for the $[\cdot]$ modality a user has to supply a loop invariant *Inv*. Intuitively, the rule establishes three things: (1) In the state before the loop is executed, some invariant *Inv* holds. (2) If the body of the loop terminates normally (there is no `break` and no exception is thrown but possibly `continue` is used) then at the end of a single execution of the loop body the invariant *Inv* has to hold again. (3) Provided *Inv* holds, the formula $\Box$ has to hold during and continuously after loop body execution in all of the following cases: (i) when the loop body is executed once and terminates normally, (ii) when the loop body is not executed (the loop condition is not satisfied), and (iii) when the loop body terminates abruptly (by `break`, `continue`, or throwing an exception) resulting in a termination of the whole loop.

Formally, the `while` rule for $[\![\cdot]\!]$ is the following:

$$\frac{\Box \vdash \mathcal{U}Inv \quad Inv \vdash \langle\Box\rangle true \quad Inv \vdash [\Box]Inv \quad Inv \vdash [\![\Box\Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\Box \texttt{ while(}a\texttt{) \{}p\texttt{\}} \Box]\!]\Box} \tag{R4}$$

where

$\Box \equiv$ `if(`a`) {`l_{break} `: {try {`l_{cont} `: {`p'`} abort;} catch(Exception e){}}}`
$\Box \equiv$ `if(`a`)` l_{cont} `:` l_{break} `: {`p'`}`

In the above rule, $\square$ is a (possibly empty) sequence "$l_1 : \triangleright\triangleright\triangleright l_n :$ " of labels, and p' is p with (a) every "`continue;`" and every "`continue` l_i`;`" changed to "`break` l_{cont}`;`" and (b) every "`break;`" and every "`break` l_i`;`" changed to "`break` l_{break}`;`". The three premisses establish the three conditions listed above, respectively. When the program p' terminates normally, the `abort` in $\square$ is reached and, thus, the formula $\langle\square\rangle$*true* evaluates to *false* and $[\square]$*Inv* has to be proved. Enclosing program p' in "`if(`a`)` $\triangleright\triangleright\triangleright$" takes care of both cases, where the loop body is executed (intermediate loop body execution) and where it is not executed (loop exit). They are later in the proof considered separately by applying the rule for `if`.

4.2 Additional Sequent Calculus Rules for Transactions

Additional Syntax. Before presenting the sequent rules for transactions, we first have to introduce some new programming constructs (statements) and transaction markers to JAVA CARD DL.

The three new statements are `bT` (JAVA CARD beginning of a transaction), `cT` (JAVA CARD end of a transaction, i.e., commit), and `aT` (JAVA CARD end of a transaction, i.e., abort). These statements are used in the proof when the transaction is started resp. finished in the JAVA CARD program. The statements are only part of the rules and not the JAVA CARD programming language. Thus for example, when a transaction is started in a JAVA CARD program by a call to `JCSystem.beginTransaction()` the calculus assumes the following implementation of `beginTransaction()`:

```
public class JCSystem {
  private static int _transDepth = 0;
  public static void beginTransaction() throws TransactionException {
    if(_transDepth > 0)
      TransactionException.throwIt(TransactionException.IN_PROGRESS);
    _transDepth++;
    bT;
  }
  ...
```

Thus, when we encounter any of `bT`, `cT` or `aT` in our proof we can assume they are properly used (nested).

The second thing we need is the possibility to mark modalities (resp. the programs they contain) with a tag saying that a transaction is in progress. We will use two kinds of tags and make them part of the inactive program prefix $\square$ in the sequent. The two markers are: "TRcommit:"—a transaction is in progress and is expected to be committed (`cT`), and "TRabort:"—a transaction is in progress and is expected to be aborted (`aT`). This distinction is very helpful in taking care of conditional assignments—since we know how the transaction is going to terminate "beforehand" we can treat conditional assignments correspondingly, commit them immediately in the first case or "forget" them in the second case.

Rules for Beginning a Transaction. For each of the three operators ($\langle\cdot\rangle$, $[\cdot]$, $[\![\cdot]\!]$) there is one "begin transaction" rule (the rules for $\langle\cdot\rangle$ and $[\cdot]$ are identical, so we only show one of them):

$$\frac{\Box \vdash \mathcal{U}\Box \quad \Box \vdash \mathcal{U}[\![\mathsf{TRcommit:}\ \Box\Box]\!]\Box \quad \Box \vdash \mathcal{U}[\![\mathsf{TRabort:}\ \Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\Box\ \mathtt{bT;}\ \Box]\!]\Box} \tag{R5}$$

$$\frac{\Box \vdash \mathcal{U}\langle\mathsf{TRabort:}\ \Box\Box\rangle\Box \quad \Box \vdash \mathcal{U}\langle\mathsf{TRcommit:}\ \Box\Box\rangle\Box}{\Box \vdash \mathcal{U}\langle\Box\ \mathtt{bT;}\ \Box\rangle\Box} \tag{R6}$$

In case of the $[\![\cdot]\!]$ operator the following things have to be established. First of all, $\Box$ has to hold before the transaction is started. Then we split the sequent into two cases: the transaction will be terminated by a commit, or the transaction will be terminated by an abort. In both cases the sequent is marked with the proper tag, so that corresponding rules can be applied later, depending on the case. The $\langle\cdot\rangle$ and $[\cdot]$ rules for "begin transaction" are very similar to $[\![\cdot]\!]$ except that $\Box$ does not have to hold before the transaction is started.

Rules for Committing and Aborting Transactions. These rules are the same for all three operators, so we only show the $[\![\cdot]\!]$ rules.

The first two rules apply when the expected type of termination is encountered ("TRcommit:" for commit resp. "TRabort:" for abort). In that case, the corresponding transaction marker is simply removed, which means that the transaction is no longer in progress. These are the rules:

$$\frac{\Box \vdash \mathcal{U}[\![\Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\mathsf{TRcommit:}\ \Box\ \mathtt{cT;}\ \Box]\!]\Box} \tag{R7}$$

$$\frac{\Box \vdash \mathcal{U}[\![\Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\mathsf{TRabort:}\ \Box\ \mathtt{aT;}\ \Box]\!]\Box} \tag{R8}$$

We also have to deal with the case where the transaction is terminated in an unexpected way, i.e., a commit is encountered when the transaction was expected to abort and vice versa. In this case we simply use an axiom rule, which immediately closes the proof branch (one of the proof branches produced by the "begin transaction" rule will always become obsolete since each transaction can only terminate by either commit or abort). The rules are the following:

$$\frac{}{\Box \vdash \mathcal{U}[\![\mathsf{TRabort:}\ \Box\ \mathtt{cT;}\ \Box]\!]\Box} \tag{R9}$$

$$\frac{}{\Box \vdash \mathcal{U}[\![\mathsf{TRcommit:}\ \Box\ \mathtt{aT;}\ \Box]\!]\Box} \tag{R10}$$

Rules for Conditional Assignment Handling within a Transaction. Finally, we come to the essence of conditional assignment handling in our rules. In case the transaction is expected to commit, no special handling is required—all the assignments are executed immediately. Thus, the rule for an assignment in the scope of $[\![\mathsf{TRcommit:}\ \triangleright\triangleright\triangleright]\!]$ is the same as the rule for an assignment within $[\cdot]$ (the same holds for all other programming constructs). Note that, even using the $[\![\mathsf{TRcommit:}\ \triangleright\triangleright\triangleright]\!]$ modality, $\Box$ only has to hold at the end of the transaction, which is considered to be atomic.

$$\frac{\Box \vdash \mathcal{U}\{loc := expr\}[\![\mathsf{TRcommit:}\ \Box\Box]\!]\Box}{\Box \vdash \mathcal{U}[\![\mathsf{TRcommit:}\ \Box\ \mathtt{loc = expr;}\ \Box]\!]\Box} \tag{R11}$$

In case a transaction is terminated by an abort, all the conditional assignments are rolled back as if they were not performed. If we know that the transaction is going to abort because of a `TRabort:` marker, we can deliberately choose not to perform the updates to persistent objects as we encounter them. However, we cannot simply skip them since the new values assigned to (fields of) persistent objects during a transaction may be referred to later in the same transaction (before the abort). The idea to handle this, is to assign the new value to a copy of the object field or array element while leaving the original unchanged, and to replace—until the transaction is aborted—references to persistent fields and array elements by references to their copies holding the new value. Note that if an object field to which no new value has been assigned is referenced (and for which therefore no copy has been initialised), the original reference is used.

Making this work in practice requires changing the assignment rule for the cases where a transaction is in progress and is expected to abort (i.e., where the "`TRabort:`" marker is present). Also the rules for update evaluation change a bit, which changes the semantics of an update as well, see description of the rule below. The following is the assignment rule for the $[\![\cdot]\!]$ modality with the "`TRabort:`" tag present. The corresponding rules for $\langle\cdot\rangle$ and $[\cdot]$ are the same:

$$\frac{\square \;\vdash\; \mathcal{U}\{loc' := expr'\}[\![\texttt{TRabort:}\,\square\,\square\,]\!]\square}{\square \;\vdash\; \mathcal{U}[\![\texttt{TRabort:}\,\square\;\texttt{loc = expr;}\;\square\,]\!]\square} \tag{R12}$$

As usual *expr* has to be a logical term. To handle objects fields persistent arrays elements, all sub-expressions such as $obj{\triangleright}a_1{\triangleright}arr[e]{\triangleright}a_2\,{\triangleright}{\triangleright}{\triangleright}$ in *expr* are replaced by $obj{\triangleright}a_1'{\triangleright}arr'[e']'{\triangleright}a_2'\,{\triangleright}{\triangleright}{\triangleright}$ in $expr'$ (for object fields the prime denotes a copy of that field and for array access function [] the prime denotes a "shadow" access function that operates on copies of elements of a given array). The first reference *obj* or *arr* (as in $arr[i]{\triangleright}a$) in *expr* is not primed, since it is either a local variable, which is not persistent, or the `this` reference, which is not assignable, or a static class reference, like `SomeClass`, which also can be viewed as not assignable. All subexpressions that are local variables are left unchanged in $expr'$. The expression *loc* on the left side of the assignment and the subexpression *e* are changed into loc' resp. e' in the same way as all the subexpressions in *expr*.

As mentioned, the semantics of an update has to be changed to take care of the cases when a copy of an object's field has not been initialised. In the new semantics, if the value of $obj{\triangleright}a'$ or $arr[i]'$ is referred to in an update but is not known (i.e., there was no such value assigned in the preceding updates) then it is considered to be equal to $obj{\triangleright}a$ or $arr[i]$, respectively.

The assignments to the copies are not visible outside the transaction, where the original values are used again—the effect of a roll-back is accomplished. Each separate transaction has to have its own copies of fields or array elements, so the second encountered transaction can, for example, use $''$, the third one $'''$, etc.

One more thing that we have to handle here is the case when the programmer explicitly defines an array to be transient (the above rule assumes that it was not the case). It is not possible to know beforehand which arrays are transient and which are not, since they are defined to be transient by reference and not

by name. This problem can be treated by adding an extra field to each array (only in the rules) indicating whether the given array is transient or persistent (rules for initialising arrays can set this field). Then for each occurrence of array reference *arr* in *loc* and *expr* in rule (R12) we can split the proof into two cases, following the schema:

$$\frac{\Box \lessdot \mathcal{U}(o \triangleright arr' \bowtie trans \stackrel{\triangleright}{=} true) \;\vdash\; \mathcal{U}\{o \triangleright arr'[e'] := expr'\}[\![\mathsf{TRabort}: \Box\Box]\!]\Box \quad \Box \lessdot \mathcal{U}(o \triangleright arr' \bowtie trans \stackrel{\triangleright}{=} false) \;\vdash\; \mathcal{U}\{o \triangleright arr'[e']' := expr'\}[\![\mathsf{TRabort}: \Box\Box]\!]\Box}{\Box \;\vdash\; \mathcal{U}[\![\mathsf{TRabort}: \Box \texttt{ o.arr[e] = expr; } \Box]\!]\Box} \quad \text{(R13)}$$

The remaining rules for $[\![\mathsf{TRabort}: \cdot]\!]$ (i.e., for other programming constructs) are the same as for $[\cdot]$, and the remaining rules for $[\mathsf{TRabort}: \cdot]$ and $\langle \mathsf{TRabort}: \cdot \rangle$ are the same as if there were no transaction marker.

5 Examples

In the following, we show two examples of proofs using the above rules. The first example shows how the $[\![\cdot]\!]$ assignment and `while` rules are used, the second example shows the transaction rules in action. The formula we are trying to prove in the second example is deliberately not provable and shows the importance of the transaction mechanism when it comes to "throughout" properties.

The proofs presented here may look like tedious work, but most of the steps can be done automatically, in fact the only place where user interaction is required, is providing the loop invariant. The KeY system provides necessary mechanisms to perform proof steps automatically whenever possible.

Example 1. Consider the program p shown on the right. We show that throughout the execution of this program, the strong invariant $\Box \equiv x \geq 2$ holds, i.e., we prove the formula $x \geq 2 \rightarrow [\![p]\!]x \geq 2$. Figure 1 shows the whole proof labelled with applied rules. Here we only point out the most interesting things.

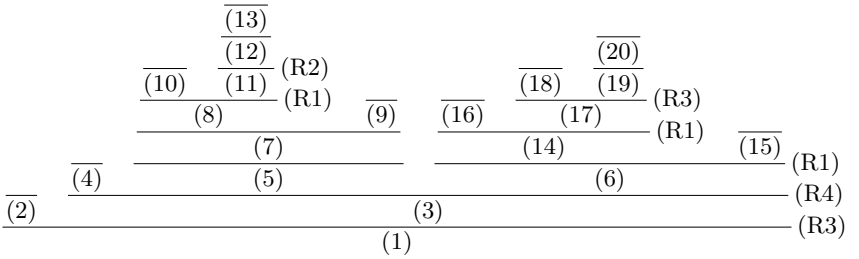
```
x = 3;
while (x < 10) {
  if(x == 2) x = 1;
  else x++;
}
```

When applying the `while` rule (R4) to (3) formula $x \geq 3$ has to be used as the loop invariant *Inv*. Using $Inv' = \Box = x \geq 2$ would not be enough, because the statement `x = 1` inside the `if` statement could not be discarded and x would be assigned 1, which would break the $x \geq 2$ property.

For $x < 10$, the `abort` statement in $\Box$ is reached after some execution steps (due to space restrictions, we do show the corresponding proof steps). Since `abort` is non-terminating, the formula $\langle$`abort;`$\rangle$*true* is false and thus (5) can be reduced to (7). All sequents with an empty modality ($[\![\cdot]\!]$ or $[\cdot]$) are reduced by removing the modality; the resulting sequents are then first-order provable. Sequents (9), (10) and (16) are valid by contradiction in the antecedent.

Example 2. Now consider the following program p (fields of o are persistent):

```
bT;
  o.x = 60;
  o.y = 40;
cT;
```



$$x \geq 2 \vdash [\![\texttt{x = 3;} \ldots]\!] x \geq 2 \quad (1)$$

$$x \geq 2 \vdash x \geq 2 \quad (2)$$

$$x \geq 2 \vdash \{x := 3\}[\![\texttt{while} \ldots]\!] x \geq 2 \quad (3)$$

$$x \geq 2 \vdash \{x := 3\} x \geq 3 \quad (4)$$

$$x \geq 3 \vdash \langle \alpha \rangle \mathit{true},\ [\texttt{if(x<10)}\lambda\{\beta\}] x \geq 3 \quad (5)$$

$$x \geq 3 \vdash [\![\texttt{if(x<10)}\lambda\{\beta\}]\!] x \geq 2 \quad (6)$$

$$x \geq 3,\ x < 10 \vdash [\texttt{if(x<10)}\lambda\{\beta\}] x \geq 3 \quad (7)$$

$$x \geq 3,\ x < 10,\ x < 10 \vdash [\lambda\{\beta\}] x \geq 3 \quad (8)$$

$$x \geq 3,\ x \geq 10,\ x < 10 \vdash [\,] x \geq 3 \quad (9)$$

$$x \geq 3,\ x < 10,\ x \doteq 2 \vdash [\texttt{x = 1;}] x \geq 3 \quad (10)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash [\texttt{x = x + 1;}] x \geq 3 \quad (11)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash \{x := x + 1\}[\,] x \geq 3 \quad (12)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x + 1 \geq 3 \quad (13)$$

$$x \geq 3,\ x < 10 \vdash [\![\lambda\{\beta\}]\!] x \geq 2 \quad (14)$$

$$x \geq 3,\ x \geq 10 \vdash [\![\,]\!] x \geq 2 \quad (15)$$

$$x \geq 3,\ x < 10,\ x \doteq 2 \vdash [\![\texttt{x = 1;}]\!] x \geq 2 \quad (16)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash [\![\texttt{x = x + 1;}]\!] x \geq 2 \quad (17)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x \geq 2 \quad (18)$$

$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash \{x := x + 1\}[\![\,]\!] x \geq 2 \quad (19)$$

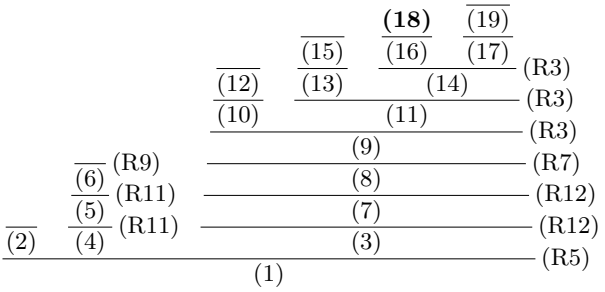
$$x \geq 3,\ x < 10,\ \neg x \doteq 2 \vdash x + 1 \geq 2 \quad (20)$$

Abbreviations: $\alpha \equiv$ `if(x < 10) {`$\ldots\beta$`; abort;`$\ldots$`}` $\quad \lambda \equiv l_{cont}\texttt{:}l_{break}\texttt{:}$
$\beta \equiv$ `if(x == 2) x = 1; else x++;`

Fig. 1. The proof from Example 1

```
t = o.x;
o.x = o.y;
o.y = t;
```

We will try to prove that the strong invariant $o{\triangleright}x + o{\triangleright}y \stackrel{\triangleright}{=} 100$ holds throughout the execution of this program. Note that this is not provable. The proof attempt is shown in Fig. 2. Again, some of the sequents are first-order provable after appropriate reductions. Notice that applying the assignment rule (R11) (resp. (R12)) inside a transaction does not branch. Sequent (6) is proved valid by the axiom rule (R9) (transaction commits unexpectedly). Sequent (18) is not provable. Inspecting our program closely shows that indeed both $o{\triangleright}x$ and $o{\triangleright}y$ are equal to 40 at some point (after `o.x = o.y;` is executed) and their sum is 80, which violates the property we wanted to prove. Thus there is one open proof goal in the proof tree.



$$o.x + o.y \doteq 100 \;\vdash\; [\![\texttt{bT; } \ldots]\!] o.x + o.y \doteq 100 \tag{1}$$

$$o.x + o.y \doteq 100 \;\vdash\; o.x + o.y \doteq 100 \tag{2}$$

$$o.x + o.y \doteq 100 \;\vdash\; [\![\mathsf{TRcommit}\texttt{: o.x = 60; } \ldots]\!] o.x + o.y \doteq 100 \tag{3}$$

$$o.x + o.y \doteq 100 \;\vdash\; [\![\mathsf{TRabort}\texttt{: o.x = 60; } \ldots]\!] o.x + o.y \doteq 100 \tag{4}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x' := 60\}[\![\mathsf{TRabort}\texttt{: o.y = 40; } \ldots]\!] o.x + o.y \doteq 100 \tag{5}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x' := 60\}\{o.y' := 40\}[\![\mathsf{TRabort}\texttt{: cT; } \ldots]\!] o.x + o.y \doteq 100 \tag{6}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}[\![\mathsf{TRcommit}\texttt{: o.y = 40; } \ldots]\!] o.x + o.y \doteq 100 \tag{7}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}[\![\mathsf{TRcommit}\texttt{: cT; } \ldots]\!] o.x + o.y \doteq 100 \tag{8}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}[\![\texttt{t = o.x; } \ldots]\!] o.x + o.y \doteq 100 \tag{9}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\} o.x + o.y \doteq 100 \tag{10}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}[\![\texttt{o.x = o.y; } \ldots]\!] o.x + o.y \doteq 100 \tag{11}$$

$$o.x + o.y \doteq 100 \;\vdash\; 60 + 40 \doteq 100 \tag{12}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\} o.x + o.y \doteq 100 \tag{13}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}[\![\texttt{o.y = t; } \ldots]\!] o.x + o.y \doteq 100 \tag{14}$$

$$o.x + o.y \doteq 100 \;\vdash\; 60 + 40 \doteq 100 \tag{15}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\} o.x + o.y \doteq 100 \tag{16}$$

$$o.x + o.y \doteq 100 \;\vdash\; \{o.x := 60\}\{o.y := 40\}\{t := o.x\}\{o.x := o.y\}\{o.y := t\}[\![\,]\!] o.x + o.y \doteq 100 \tag{17}$$

$$o.x + o.y \doteq 100 \;\vdash\; 40 + 40 \doteq 100 \tag{18}$$

$$o.x + o.y \doteq 100 \;\vdash\; 40 + 60 \doteq 100 \tag{19}$$

Fig. 2. The proof from Example 2

6 Conclusions and Future Work

We introduced the "throughout" modality (and, thus, strong invariants) to JAVA CARD Dynamic Logic and presented the necessary sequent calculus rules to handle this modality and conditional assignments in JAVA CARD transactions. Intro-

duction of this modality was a manageable task and the set of presented rules is quite easy to use in theorem proving as shown in the examples. Our future plan is to implement our rules in the KeY prover and then try our calculus with "real" examples.

References

1. W. Ahrendt, T. Baar, B. Beckert, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system: Integrating object-oriented design and formal methods. In R.-D. Kutsche and H. Weber, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 327–330. Springer, 2002.
2. B. Beckert. A dynamic logic for the formal verification of JavaCard programs. In I. Attali and T. Jensen, editors, *Revised Papers, Java on Smart Cards: Programming and Security, Cannes, France*, LNCS 2041, pages 6–24. Springer, 2001.
3. B. Beckert and B. Sasse. Handling Java's abrupt termination in a sequent calculus for Dynamic Logic. In B. Beckert, R. France, R. Hähnle, and B. Jacobs, editors, *Proceedings, IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development, Siena, Italy*, pages 5–14. TR DII 07/01, Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Siena, 2001.
4. B. Beckert and S. Schlager. A sequent calculus for first-order dynamic logic with trace modalities. In R. Gorè, A. Leitsch, and T. Nipkow, editors, *Proceedings, International Joint Conference on Automated Reasoning, Siena, Italy*, LNCS 2083, pages 626–641. Springer, 2001.
5. J. C. Bradfield, J. K. Filipe, and P. Stevens. Enriching OCL using observational mu-calculus. In R.-D. Kutsche and H. Weber, editors, *Proceedings, Fundamental Approaches to Software Engineering (FASE), Grenoble, France*, LNCS 2306, pages 203–217. Springer, 2002.
6. Z. Chen. *Java Card Technology for Smart Cards.* Addison Wesley, 2000.
7. D. Harel. Dynamic Logic. In D. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic, Volume II: Extensions of Classical Logic.* Reidel, 1984.
8. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
9. KeY project homepage. `http://i12www.ira.uka.de/~projekt/`.
10. D. Kozen and J. Tiuryn. Logic of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 14, pages 89–133. Elsevier, 1990.
11. W. Mostowski. Rigorous development of JavaCard applications. In T. Clarke, A. Evans, and K. Lano, editors, *Proc. Fourth Workshop on Rigorous Object-Oriented Methods, London*, 2002.
    `http://www.cs.chalmers.se/~woj/papers/room2002.ps.gz`.
12. V. R. Pratt. Semantical considerations on Floyd-Hoare logic. In *Proceedings, 18th Annual IEEE Symposium on Foundation of Computer Science*, 1977.
13. Sun Microsystems, Inc. *Java Card 2.2 Application Programming Interface*, 2002.
14. Sun Microsystems, Inc. *Java Card 2.2 Runtime Environment Specification*, 2002.
15. Sun Microsystems, Inc. *Java Card 2.2 Virtual Machine Specification*, 2002.
16. K. Trentelman and M. Huisman. Extending JML specifications with temporal logic. In *Algebraic Methodology And Software Technology (AMAST '02)*, LNCS 2422, pages 334–348. Springer-Verlag, 2002.

Monad-Independent Hoare Logic in HasCasl

Lutz Schröder and Till Mossakowski

BISS, Department of Computer Science, University of Bremen

Abstract. Monads have been recognized by Moggi as an elegant device for dealing with stateful computation in functional programming languages. It is thus natural to develop a Hoare calculus for reasoning about computational monads. While this has previously been done only for the state monad, we here provide a generic, monad-independent approach, which applies also to further computational monads such as exceptions, input/output, and non-determinism. All this is formalized within the logic of HasCasl, a higher-order language for functional specification and programming. Combination of monadic features can be obtained by combining their loose specifications. As an application, we prove partial correctness of Dijkstra's nondeterministic version of Euclid's algorithm in a monad with nondeterministic dynamic references.

1 Introduction

One of the central concepts of modern functional programming is the encapsulation of side effects via monads following the seminal paper [11]. In particular, state monads are used to emulate an imperative programming style in the functional programming language Haskell [15]. Monads can be used to abstract from a particular notion of computation, since they model a wide range of computational effects: e.g., stateful computations, non-determinism, partiality, exceptions, input, and output can all be viewed as monadic computations, and so can various combinations of these concepts such as non-deterministic stateful computations.

Here, we show how one can also build a generic *logical* environment for reasoning about generic monadic computations by providing a monad-independent Hoare calculus. These results are developed in the framework of the algebraic specification language HasCasl. In this way, we generalize the suggestions of [11], which were aimed purely at a state monad with state interpreted as global store.

HasCasl has been introduced in [14] as a higher order extension of the first order algebraic specification language Casl [3]. HasCasl is geared towards specification of functional programs, in particular in Haskell; in fact, HasCasl has an executable subset that corresponds quite closely to a large subset of Haskell. Features of HasCasl include partial and total higher order types, polymorphism, type classes, and general recursive functions. The technical requirement for a general treatment of monads is support for constructor classes, which are a straightforward extension of HasCasl's type classes. In the correspondingly

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 261–277, 2003.

extended language, one can easily specify not only the operations, but also the axioms associated to a monad.

Using HasCasl's internal logic, one can give a semantics to Hoare triples independently of the internal structure of the monad. The HasCasl logic then provides a Hoare calculus that allows reasoning about partial correctness as well as loosely specifying imperative programs. We provide both a generic kernel calculus and specialized calculi that provide additional rules dealing with monad-specific operations such as assignment. We end up with an environment that offers not only a combination of functional and imperative programming (as provided in Haskell), but also a surrounding logic that is rather effortlessly adapted to the specification of both functional and imperative aspects.

2 HasCasl

The language HasCasl has been introduced in [14] as a higher order extension of Casl, based on the partial □-calculus. We give a brief summary of how basic HasCasl specifications are written and what they mean. For more details on both syntax and semantics, see [14].

Any HasCasl specification determines essentially two things: a *signature* consisting of classes, types, and operations, and associated *axioms* that the operations are required to satisfy. Basic types are introduced by means of the keyword **type**. Types may be parametrized by type arguments; e.g., we may write

var $a : Type$
type $List\ a$

and obtain a unary type constructor *List*. There are built-in type constructors (with fixed interpretations) $\_ \times \_$, $\_ \times \_ \times \_$ etc. for product types, $\_ \to ? \_$ and $\_ \to \_$ for partial and total function types, respectively, $Pred\ \_$ for predicate types, and a unit type *Unit*. In particular, the function types of Haskell are really partial function types.

Next, an *operator* is a constant of some type, declared by

op $f : t$

where t is a type. Since types may contain type variables, operators can be polymorphic in the style of ML.

From the given operators, we may form higher order terms in the usual way: a term is either a variable, an application, a tuple, or a □-abstraction. Such terms may then be used in *axioms* formulated, to begin, in what we shall call the *external logic*. This external logic offers the usual logical connectives (conjunction, negation etc.) as well as universal and existential quantifiers, where the outermost universal quantifications may also be over type variables, strong and existential equality denoted by $=$ and $\overset{e}{=}$, respectively, and definedness assertions $def\ \square$ (the latter feature and the distinction between the various equalities are related to partial functions; cf. [12] for a detailed discussion). The notation used in the examples below is largely self-explanatory, so we shall refrain from listing the syntactic details here. It is important to note that formulas of the external

logic, including external equations, are not regarded as terms of a program and hence cannot be λ-abstracted. Partial functions are, unlike in Haskell, required to be strict; non-strict functions can be emulated by means of the procedural lifting method, for which suitable syntactical sugar is provided.

The semantics of a HasCasl specification is the class of its (set-theoretic) *intensional Henkin models*: a function type need not contain all set-theoretic functions, and two functions that yield the same value on every input need not be equal; see [14] for a discussion of the rationale behind this. If desired, extensionality of models may be forced by means of an axiom expressible within the language.

A consequence of the intensional semantics is the presence of an intuitionistic *internal logic* that lives within λ-terms. One can specify an *internal equality* (for which the symbol $=$ is built-in syntactical sugar) to be used within λ-terms, which then allows *specifying* the full set of logical operations and quantifiers of intuitionistic logic; this is carried out in detail in [14]. There is built-in syntactical sugar for the internal logic, invoked by means of the keyword **internal** which signifies that formulas in the following block are to be understood as formulas of the internal logic.

By means of the internal logic, one can then specify a class of complete partial orders and fixed point recursion in much the same style as in HOLCF [13]. On top of this, syntactical sugar is provided that allows recursive function definitions in the style used in functional programming, indicated by the keyword **program**. Similarly, the no-junk-no-confusion axioms associated to datatypes are implicitly coded by means of the internal logic.

HasCasl supports *type classes*. These are declared in the form

class C

and are to be understood as subsets of the syntactical universe of all types. Types as well as type variables can be restricted to belong to an assigned class, e.g. by writing

type $t : C$

In particular, axioms and operators may be polymorphic over classes. Classes may be subclasses of each other, and they may have generic instances. By attaching polymorphic operators and axioms to a class, one achieves a similar effect as with Haskell's type classes.

In a similar vein, one can add *constructor classes* to HasCasl. They can be interpreted as predicates on the syntactical universe of abstracted type expressions (also called *pseudotypes*), e.g.

$$\lambda\, a : Type \bullet a \to ?\ List\ a$$

As for type classes, there are constructor subclasses; types, operators, axioms may be polymorphic over constructor classes; and this polymorphism is semantically coded by collections of instances. A typical example of a constructor class is the class of monads (see Fig. 1); an example of a constructor subclass can be found in Fig. 3.

In summary, HasCasl is a language that allows both property-oriented specification and functional programming; executable HasCasl specifications may easily be translated into Haskell programs.

3 Monads for Computations

On the basis of the seminal paper [11], monads are being used for encapsulating side effects in modern functional programming languages; in particular, this idea is one of the central concepts of Haskell [8]. Intuitively, a monad associates to each type A a type TA of computations of type A; a function with side effects that takes inputs of type A and returns values of type B is, then, just a function of type $A \to TB$. This approach abstracts away from particular notions of computation such as store, non-determinism, non-termination etc.; a surprisingly large amount of reasoning can in fact be carried out independently of the choice of such a notion.

More formally, a monad on a given category $\mathbf{C}$ can be defined as a *Kleisli triple* $(T, \eta, \_^*)$, where $T : \mathrm{Ob}\,\mathbf{C} \to \mathrm{Ob}\,\mathbf{C}$ is a function, the *unit* η is a family of morphisms $\eta_A : A \to TA$, and $\_^*$ assigns to each morphism $f : A \to TB$ a morphism $f^* : TA \to TB$ such that

$$\eta_A^* = id_{TA} \qquad f^*\eta_A = f \qquad \text{and} \qquad g^*f^* = (g^*f)^*$$

This description is equivalent to the more familiar one via an endofunctor with unit and multiplication [10]. A monad gives rise to a *Kleisli category* over $\mathbf{C}$, which has the same objects as $\mathbf{C}$ and 'functions with side effects' $f : A \to TB$ as morphisms from A to B; the composite of two such functions g and f is just g^*f. This composite will also be denoted $f; g$.

In order to support a language with finitary operations and multi-variable contexts (see below), one needs a further technical requirement: a monad is called *strong* if it is equipped with a natural transformation

$$t_{A,B} : A \times TB \to T(A \times B)$$

called *tensorial strength*, subject to certain coherence conditions (see e.g. [11]); this is equivalent to enrichment of the monad over $\mathbf{C}$ (see discussion and references in [11]).

Example 1 ([11]). Computationally relevant monads on **Set** (since strength is equivalent to enrichment, all monads on **Set** are strong) include

- stateful computations with possible non-termination: $TA = (S \to ? (A \times S))$, where S is a fixed set of states and $\_ \to ? \_$ denotes the partial function type;
- (finite) non-determinism: $TA = \mathcal{P}_{fin}(A)$, where $\mathcal{P}_{fin}$ denotes the finite power set functor;
- exceptions: $TA = A + E$, where E is a fixed set of exceptions;
- interactive input: TA is the smallest fixed point of $X \mapsto A + (U \to X)$, where U is a set of input values.

- non-deterministic stateful computations: $TA = (S \to \mathcal{P}_{fin}(A \times S))$, where, again, S is a fixed set of states;

Figure 1 shows a specification of monads in HasCasl. As an example of an instance for this type class, a specification of the state monad is shown in Fig. 2. Since the operations of the monad are functions in the model, the monads thus specified are automatically strong, strength being equivalent to enrichment. The notation is (almost) identical to the one used in Haskell, i.e. the unit is denoted by *ret*, and the type constructor by m; the operator $\_ >>= \_$ denotes, in the above notation, the function $(x, f) \mapsto f^*(x)$. This specification is the basis for a built-in sugaring in the form of a Haskell-style do-notation: for monadic expressions e_1 and e_2,

$$\text{do } x \leftarrow e_1;\ e_2$$

abbreviates $e_1 >>= \lambda x \bullet e_2$. A slight complication concerning the axiomatization arises from the fact that partial functions are involved. Note that the first axiom has been equipped with a definedness guard. This ensures that standard monads such as the state monad with its usual definition (cf. Fig. 2 and the recent discussion on [7]) are actually subsumed, while leaving the essence of the proposed calculus untouched.

```
spec  MONAD = INTERNALLOGIC then
      class  Monad : Type → Type {
      vars   m : Monad;  a, b, c : Type
      ops    _ >>= _ : m a → (a →? m b) →? m b;
             ret : a → m a                          }
      internal {
      forall x, y : a;  y : m a;  f : a →? m b;  g : b →? m c
             • def (f x) ⇒ ((ret x) >>= f) = f x
             • (y >>= ret) = y
             • ((y >>= f) >>= g) = (y >>= (λx : a • f x >>= g))    }
```

Fig. 1. The constructor class of monads

Reasoning about a category **C** equipped with a strong monad is greatly facilitated by the fact that proofs can be conducted in an *internal language* introduced in [11]. The crucial features of this language are

- A type operator T; terms of type TB for some B are called *programs*;
- an polymorphic operator ret : $A \to TA$ corresponding to the unit;
- a binding construct, which we here denote in Haskell's do style instead of by let: terms of the form

$$\text{do } x \leftarrow p;\ q$$

```
spec  STATE [type S] = MONAD then
      type instance ST : Monad
      vars  a, b : Type
      type  ST a := S →? (a × S)
      internal {
      forall x : a; y : ST; f : a →? ST b
            • ret x = λ s : S • (x, s)
            • (y >>= f) = λ s1 : S • let (z, s2) = y s1 in f z s2    }
```

Fig. 2. Specification of the state monad

are interpreted by means of the tensorial strength and Kleisli composition (See [11] for details. This is essentially equivalent the do-notation introduced above.). Intuitively, do $x \leftarrow p;\ q$ computes p and passes the results on to q. Nested do expressions like do $x \leftarrow p;$ do $y \leftarrow q;\ \triangleright\!\triangleright\!\triangleright$ may also be denoted do $x \leftarrow p; y \leftarrow q;\ \triangleright\!\triangleright\!\triangleright$. Repeated nestings such as do $x_1 \leftarrow p_1 \mathbin{‹\triangleright\!\triangleright\!\triangleright‹} x_n \leftarrow p_n;\ q$ are somewhat inaccurately denoted in the form do $\bar{x} \leftarrow \bar{p};\ q$. Term fragments of the form $\bar{x} \leftarrow \bar{p}$ are called *program sequences*. Variables x_i that do not appear lateron may be omitted from the notation.

Terms are generally formed in a context $\Box = (x_1 : s_1 \mathbin{‹\triangleright\!\triangleright\!\triangleright‹} x_n : s_n)$ of variables with assigned types. Thanks to an equivalence theorem proved in [11], this language (with further term formation rules and a deduction system) can be used both in order to define morphisms in **C** and in order to prove equalities between them. For example, morphisms $f : A \to TB$ and $g : B \to TC$ may also be seen as terms $f : TB$ and $g : TC$ in context $x : A$ and $y : B$, respectively; the Kleisli composite $f; g$ is represented by do $y \leftarrow f;\ g$.

On top of a monad, one can generically define control structures such as a while loop. However, such definitions require general recursion, which is realized in HASCASL by means of fixed point recursion on cpos. Thus, one has to restrict to monads that allow lifting a cpo structure on A to a cpo structure on the type TA of computations in such a way that the monad operations become continuous. This is an example of a *constructor subclass*; the corresponding specification of *cpo-monads* is shown in Fig. 3. Function types indicated by $\xrightarrow{c}$ indicate types of continuous functions [14]. The relevant examples including the ones given above belong to this subclass.

As an example of a recursively defined control structure we introduce an iteration construct which generalizes the while loop by extending it with a default return value (the while loop as programmed e.g. in the Haskell prelude returns only a unit value) which is fed through the iteration. This has the advantage that the construct makes sense also for 'stateless' monads; e.g., iteration in the nondeterminism monad results in a function that has all values as outcomes that can be reached by repeatedly applying the original function while a given condition holds. The (executable) specification of the iteration construct is shown in Fig. 4. Note that the while loop is just iteration ignoring the return value.

```
spec CpoMonad = Recursion and Monad then
     class CpoMonad < Monad {
     vars m : CpoMonad; a : Cpo
     type m a : Cpo
     ops _ >>= _ _ : m a ->c (a ->c? m b) ->c? m b;
         return : a ->c m a                          }
```

Fig. 3. The constructor subclass of cpo-monads

```
spec Iteration = CpoMonad and Bool then
     vars m : CpoMonad; a : Cpo
     op iter : (a ->c m Bool) ->c (a ->c? m a) ->c a ->c? m a
     program iter test f x =
                 do b <- test x
                    if b then
                         do y <- f x
                            iter test f y
                         else return x
     op while(b : m Bool)(p : m Unit) : m Unit = iter (\x . b) (\x . p) ()
```

Fig. 4. The iteration control structure

4 The Generic Hoare Calculus

We now proceed to describe a Hoare-calculus which is generic over the underlying monad. (Similar calculi discussed in [5,11] are specific for the state monad, where 'state' is additionally restricted to mean global store). We shall be using the notation for monads introduced in the previous section (T, $\square$ etc.) throughout, as well as the internal language discussed at the end of the previous section.

As usual, the calculus will be concerned with proving *Hoare triples* consisting of a stateful expression together with a pre- and a postcondition. Since in general we cannot undo changes to the 'state', we have to require the pre- and postconditions to 'leave the state unchanged' in a suitable sense in order to guarantee composability of Hoare triples, at the same time allowing the conditions to read the state.

Definition 2. A program p is called *side-effect free* if

$$(\text{do } y \leftarrow p;\ \text{ret} *) = \text{ret} * \qquad (\text{shorthand: } \mathit{sef}(p))$$

where $*$ is the unique element of the unit type.

Note that $\mathit{sef}(p)$ implies that p is always defined. Properties such as side-effect freeness are said to hold for a program sequence iff they hold for each of the component programs.

Lemma 3. *If p is side-effect free, then*

$$(\text{do } x \leftarrow p;\ q) = q$$

for each program q that does not contain x.

Example 4. A program p is side-effect free

- in the state monad iff p terminates and does not change the state;
- in the non-determinism monad iff p always has at least one possible outcome;
- in the exception monad iff p terminates normally;
- in the interactive input monad iff p never reads any input;
- in the non-deterministic state monad iff p does not change the state and always has at least one possible outcome (i.e. never gets stuck).

A program p is called *stateless* if it factors through η, i.e. if it is just a value inserted into the monad ('*p exists*' in the terminology of [11]) – otherwise, it is called *stateful*. E.g. in the state monad, p is stateless iff it neither changes nor reads the state. Stateless programs are side-effect free, but not vice versa.

We will want to regard programs that return truth values as formulas with side effects. We equip such formulas with a notion of global validity, denoted explicitly by a 'global box' $\boxed{G}$:

Definition 5. Given a term t of type $T\Omega$, where Ω denotes the type of internal truth values, $\boxed{G}t$ abbreviates

$$t = \text{do } x \leftarrow t;\ \text{ret}\,\top$$

read as a formula of the internal logic.

If t is side-effect free, then $\boxed{G}t$ simplifies to $t = \text{ret}\,\top$; otherwise, the formula above ensures that the right hand side has the same side-effect as t.

Remark 6. Note that the equality in the definition of $\boxed{G}t$ above is strong equality. In particular, in the classical case $\boxed{G}t$ is true if t is undefined.

Example 7. In the monads of Example 1, $\boxed{G}t$ amounts to the following:

- in the state monad: successful execution of t from any initial state yields $\top$;
- in the non-determinism monad: t yields at most the value $\top$ (or none at all)
- in the exception monad: t yields $\top$ whenever it terminates normally.
- in the interactive input monad: the value eventually produced by t after some combination of inputs is always $\top$;
- in the non-deterministic state monad: execution of t from any initial state yields at most the value $\top$.

For meta-proofs about the Hoare logic, we require an auxiliary calculus (Fig. 5) for judgements of the form $[\bar{x} \leftarrow \bar{p}]_G\, \square$, which intuitively state that the formula $\square : \square$, which may contain $\bar{x}$, holds after $\bar{x} \leftarrow \bar{p}$. The idea is to shove all state-dependence to the outside, so that the usual logical rules apply to the remaining part. Formally, $[\bar{x} \leftarrow \bar{p}]_G\, \square$ abbreviates

$$\boxdot\ \text{do}\ \ \bar{x} \leftarrow \bar{p};\ \ \text{ret}\, \square \triangleright$$

The set of free variables of p is denoted by $FV(p)$. The calculus is sound:

Theorem 8. *If* $[\bar{y} \leftarrow \bar{q}]_G\, \square$ *is deducible from* $[\bar{x} \leftarrow \bar{p}]_G\, \square$ *by the rules of Fig. 5, then* $([\bar{x} \leftarrow \bar{p}]_G\, \square) \Rightarrow ([\bar{y} \leftarrow \bar{q}]_G\, \square)$ *holds in the internal logic.*

$$\textbf{(mp)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi_i,\ i = 1, \ldots, n \qquad \phi_1 \wedge \ldots \wedge \phi_n \Rightarrow \psi}{[\bar{x} \leftarrow \bar{p}]_G\, \psi} \qquad \textbf{(eq)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi \Rightarrow q_1 = q_2}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q_1; \bar{z} \leftarrow \bar{r}]_G\, \phi \Rightarrow \psi}$$

$$\textbf{(app)}\ \frac{[\bar{x} \leftarrow \bar{p}]_G\, \phi \qquad y \notin FV(\phi)}{[\bar{x} \leftarrow \bar{p}; y \leftarrow q]_G\, \phi} \qquad \textbf{(pre)}\ \frac{[\bar{y} \leftarrow \bar{q}]_G\, \phi \qquad x \notin FV(\phi)}{[x \leftarrow p; \bar{y} \leftarrow \bar{q}]_G\, \phi} \qquad (\eta)\ \frac{}{[x \leftarrow \text{ret}\, a]_G\, x = a}$$

$$\textbf{(ctr)}\ \frac{[\ldots; x \leftarrow p; y \leftarrow q; \bar{z} \leftarrow \bar{r}]_G\, \phi}{[\ldots; y \leftarrow (\text{do}\ \ x \leftarrow p;\ \ q); \bar{z} \leftarrow \bar{r}]_G\, \phi}\ (x \notin FV(\phi) \cup FV(\bar{r}))$$

Fig. 5. The auxiliary calculus

We can now define and give a semantics to Hoare triples:

Definition 9. A *Hoare triple*, written $\{\square\}\ \bar{x} \leftarrow \bar{p}\ \{\square\}$, consists of a program sequence $\bar{x} \leftarrow \bar{p}$, a precondition $\square : T\square$, and a postcondition $\square : T\square$ (which may contain $\bar{x}$), where $\square$ and $\square$ are side-effect free. This abbreviates the formula

$$[a \leftarrow \square; \bar{x} \leftarrow \bar{p}; b \leftarrow \square]_G\, a \Rightarrow b \triangleright$$

The fact that Hoare triples as just defined mention program *sequences* (rather than just programs) reflects the need to actually reason about results of computations, including intermediate results, as opposed to just about state changes as in the traditional case.

Example 10. A Hoare triple $\{\square\}\ x \leftarrow p\ \{\square\}$ holds

- □ in the state monad iff, whenever □ holds in a state s, then □ holds for x after successful execution of p from s with result x;
- □ in the non-determinism monad iff, whenever □ holds possibly, then □ holds for all possible results x of p;
- □ in the exception monad iff, whenever □ holds and p terminates normally, returning x, then □ holds for x;
- □ in the interactive input monad iff, whenever □ holds and p returns x after reading some sequence of inputs, then □ holds for x.
- □ in the non-deterministic state monad iff, whenever □ holds possibly in a state s, then □ holds after execution of p for all possible results x.

Remark 11. It is clear that the main application domain of Hoare triples are monads where some sort of state is involved that gives meaning to notions of 'before' and 'after'. However, as can be seen in the combination of the state monad with non-determinism, considering Hoare-triple for 'stateless' monads does make sense inasmuch as it provides for a separation of concerns.

Lastly, one can capture determinacy at least for side-effect free programs:

De□nition 12. A side-effect free program p is *deterministically side-effect free* if

$$[x \leftarrow p; y \leftarrow p]_G\, x = y \qquad (\text{shorthand: } \mathit{dsef}(p))\triangleright$$

Stateless programs are deterministically side-effect free. In most of the running examples, all side-effect free programs are deterministically side-effect free, with the unsurprising exception of the monads where non-determinism is involved. In these cases, a side-effect free program is deterministically side-effect free iff it is deterministic.

Having defined an interpretation of Hoare triples as formulas in the internal logic of HasCasl, we can now proceed to establish a set of monad-independent Hoare rules as shown in Fig. 6; the rules are lemmas in the internal logic.

The rule (wk) uses the notation □ $\Rightarrow_T$ □. This is just syntactic sugar for the Hoare triple {□} {□}; hence, (wk) is really a special case of the sequential rule (seq). The decoding of □ $\Rightarrow_T$ □ can be simplified to

$$(\text{do } a \leftarrow \square\langle b \leftarrow \square;\ \text{ret}\, a \Rightarrow b) = \text{ret}\, \top\triangleright$$

The rule (dsef) applies in particular to stateless programs $p = \text{ret}\, a$, for which the postcondition simplifies to $x = a$. Although the classical Hoare calculus does not require the usual introduction and elimination rules for logical connectives, such rules are sometimes convenient (see the example below); we have included introduction rules for conjunction and disjunction. Here, □ $\wedge$ □ abbreviates

$$\text{do } a \leftarrow \square; b \leftarrow \square;\ \text{ret}\, a \wedge b\langle$$

similarly for other logical connectives.

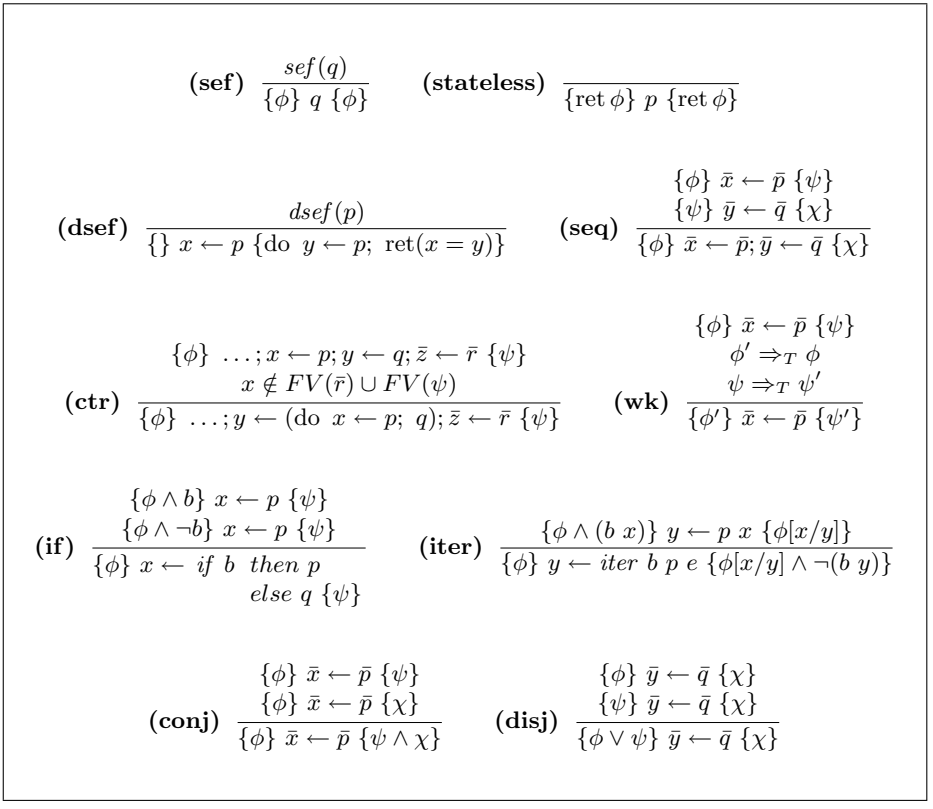


Fig. 6. The generic Hoare calculus

One typical Hoare rule that is missing here is the assignment rule; this rule only makes sense in a more specialized context where some sort of store is present. An example of an extension of the calculus by specialized rules for a particular monad is presented below.

As examples of rules for user-defined generic control-structures, we have included an invariant rule for the iteration construct introduced in Section 3 and a rule for an if-then-else construct defined in the obvious way; rules for similar control structures such as while work as usual. In the pre- and postconditions, boolean values b are implicitly converted to as $b = \mathit{true}$, and *formulas of type are implicity cast to T* via ret when needed.

The rules of the calculus are sound for arbitrary (cpo-)monads:

Theorem 13. *If a Hoare triple is derivable in a cpo-monad (monad) by the rules of Fig. 6 (excluding (iter)), then the corresponding formula is derivable in the internal language.*

It is clear that completeness can only be expected in combination with suitable monad-specific rules; e.g., the calculus becomes the usual (complete) Hoare cal-

culus when extended with an assignment rule specific to the store monad. In this sense, the calculus may be regarded as a generic framework for computational deduction systems.

5 Example: Reasoning about Dynamic References

We now apply the general machinery developed so far to the (slightly extended) domain of the classical Hoare calculus, namely states consisting of creatable and destructively updatable references (note that this is just one example of a state monad), later to be extended by non-determinism.

The reference monad R uses a type constructur Ref, where $Ref\ a$ is the set of references to values of type a. $R\ a$ is, then, the type of reference computations over a. The monad comes with operations for reading from and writing to references (besides the usual monad operations); see Fig. 7. We use the shorthand $\square(*r)$ for do $x \leftarrow read\ r$; $\square(x)$. Note that with this notation, $\mathrm{ret}(x = *r)$ is *not* stateless. Also note the difference between $\mathrm{ret}(r = s)$ (equality of references, a stateless formula) and $\mathrm{ret}(*r = *s)$ (equality of contents, a stateful formula). Moreover, recall that ret is inserted implicitly where needed.

spec REFERENCE = CPOMONAD **then**
  var $a : Cpo$
  types $R : CpoMonad$; $Ref\ a : Flatcpo$
  ops $read : Ref\ a \xrightarrow{c} R\ a$;
    $\_\_ := \_\_ : Ref\ a \xrightarrow{c} a \xrightarrow{c} R\ Unit$
forall $x, y : a$; $r, s : Ref\ a$
- $dsef(read\ r)$ *%(dsef-read)%*
- $\{\}\ r := x\ \{x = *r\}$ *%(read-write)%*
- $\{\neg r = s \wedge x = *r\}\ s := y\ \{x = *r\}$ *%(read-write-other)%*

spec DYNAMICREFERENCE = REFERENCE **then**
  var $a, b : Type$
  op $new : a \xrightarrow{c} R(Ref\ a)$
forall $x, y : a$; $r : Ref\ a$; $p : R\ b$
- $\{\}\ r \leftarrow new\ x\ \{x = *r\}$ *%(read-new)%*
- $\{x = *r\}\ s \leftarrow new\ y\ \{\neg r = s \Rightarrow x = *r\}$ *%(read-new-other)%*
- $\{\}\ r \leftarrow new\ x;\ p;\ s \leftarrow new\ y\ \{\neg r = s\}$ *%(new-distinct)%*

Fig. 7. Specification of the reference and the dynamic reference monad

The axiomatization provides all that is really necessary in order to reason about references, i.e. one does not need to rely on a particular implementation: rule *dsef-read* states that reading is deterministically side-effect free. *read-write* says that after writing to a reference, we can read the value. By contrast, writing

to a reference does not change the values of *other* references (*read-write-other*). Note that nothing is said about the nature of references; they could e.g. be integers. The specification of *dynamic* references additionally provides an operation *new* for dynamically creating new references. *read-new* states that after initializing a reference, we can read the initial value. Moreover, creation of new references does not change the values of other references (*read-new-other*). Finally, two newly created references are distinct (*new-distinct*). Note that we do not say anything about reading from references that have not been created yet.

Starting from this axiomatization, properties such as

$$\{\}\ r \leftarrow new\ x; s \leftarrow new\ y\ \{\neg r = s \wedge x = *r \wedge y = *s\} \quad (1)$$

are easily established using the Hoare rules of Fig. 6.

Another example is the nondeterminism monad, shown in Fig. 8. While *fail* yields no result and hence everything follows from it, *chaos* yields any result and hence nothing can be said about it. $[]$ is nondeterministic choice (i.e. takes the union of value sets), and *sync* synchronises two nondeterministic values (i.e. takes the intersection of value sets).

spec Nondeterminism = CpoMonad **then**
  var a : *Cpo*
  ops *fail*, *chaos* : $N\ a$;
    $\_\_[]\_\_, \_\_sync\_\_ : N\ a \xrightarrow{c} N\ a \xrightarrow{c} N\ a$
forall $x : a;\ p, q : N\ a;\ \varphi, \psi : N\Omega;\ \chi_1, \chi_2 : a \to N\Omega$
• $\{\}\ fail\ \{\psi\}$ *%(fail)%*
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p[]q\ \{\chi_1 x \vee \chi_2 x\}$ *%(join)%*
• $\{\varphi\}\ x \leftarrow p\ \{\chi_1 x\}\ \wedge\ \{\varphi\}\ x \leftarrow q\ \{\chi_2 x\}\ \Rightarrow$
  $\{\varphi\}\ x \leftarrow p\ sync\ q\ \{\chi_1 x \wedge \chi_2 x\}$ *%(sync)%*

Fig. 8. The nondeterminism monad

One advantage of the looseness of the specifications introduced so far is that we now can combine the specification of references and of nondeterminism and get a specification of nondeterministic reference computations (Fig. 9).

As an example, we prove the partial correctness of Dijkstra's nondeterministic version of Euclid's algorithm for computing the greatest common divisor [4] within this monad. Let *euclid* be the program sequence (over $NR\ Int$)

$r \leftarrow new\ x;$
$s \leftarrow new\ y;$
$while\ ret(\neg *r == *s)$
  $(if\ ret(*r > *s)\ then\ r := *r - *s\ else\ fail$
  $[]$
  $if\ ret(*s > *r)\ then\ s := *s - *r\ else\ fail)$

spec NondeterministicDynamicReference =
DynamicReference **with** $R \mapsto \neg NR$
and Nondeterminism **with** $N \mapsto \neg NR$

Fig. 9. The nondeterministic dynamic reference monad

Assuming that we have some specification of arithmetic, including *gcd* specified to be the greatest common divisor function, we now will try to prove

$$\{\}\ euclid\ \{*r = gcd(x^{\circ}\, y)\}\triangleright$$

We proceed as follows. Using (dsef), (sef), (seq), (stateless) and (conj), we can show

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y) \wedge {}^*r > {}^*s\} \\ u \leftarrow read\ r; v \leftarrow read\ s \\ \{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y) \wedge {}^*r > {}^*s \wedge u = *r \wedge v = *s\}\triangleright\end{array}$$

By arithmetic reasoning and (wk), we obtain

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y) \wedge {}^*r > {}^*s\} \\ u \leftarrow read\ r; v \leftarrow read\ s \\ \{\neg r = s \wedge gcd(u^{\circ}\, v) = gcd(x^{\circ}\, y) \wedge u > v \wedge v = *s\}\triangleright\end{array} \tag{3}$$

By (stateless), (read-write), (read-write-other), and (conj), we can show

$$\begin{array}{l}\{\neg r = s \wedge gcd(u^{\circ}\, v) = gcd(x^{\circ}\, y) \wedge u > v \wedge v = *s\} \\ r := u - v \\ \{\neg r = s \wedge gcd(u^{\circ}\, v) = gcd(x^{\circ}\, y) \wedge u > v \wedge v = *s \wedge u - v = *r\}\triangleright\end{array}$$

By arithmetic reasoning and (wk), we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(u^{\circ}\, v) = gcd(x^{\circ}\, y) \wedge u > v \wedge v = *s\} \\ r := u - v \\ \{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y)\}\triangleright\end{array}$$

By (seq) with (3) and noting that $r := {}^*r - {}^*s$ is shorthand for $u \leftarrow read\ r; v \leftarrow read\ s; r := u - v$, we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y) \wedge {}^*r > {}^*s\} \\ r := {}^*r - {}^*s \\ \{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y)\}\triangleright\end{array}$$

By (fail), we have

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y) \wedge \neg {}^*r > {}^*s\} \\ fail \\ \{\neg r = s \wedge gcd(*r^{\circ} *s) = gcd(x^{\circ}\, y)\}\triangleright\end{array}$$

Hence by (if)

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\\ \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\end{array}$$

In an entirely analogous way, we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\triangleright\end{array}$$

From these, together with (join) and (wk), we get

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\\ \quad \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail}\\ \{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\triangleright\end{array}$$

Applying (wk) and (iter) leads to

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\\ \mathit{while}\ \ \mathrm{ret}(\neg *r == *s)\\ (\ \ \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail}\\ \ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail})\\ \{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y) \wedge *r == *s\}\triangleright\end{array}$$

Using the arithmetic fact that $gcd(x^{\mathsf{c}}\, x) = x$, by (wk) we obtain

$$\begin{array}{l}\{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}\\ \mathit{while}\ \neg *r == *s\\ (\ \ \mathit{if}\ \ \mathrm{ret}(*r > *s)\ \mathit{then}\ r := *r - *s\ \mathit{else}\ \mathit{fail} \qquad (4)\\ \ [\!]\ \mathit{if}\ \ \mathrm{ret}(*s > *r)\ \mathit{then}\ s := *s - *r\ \mathit{else}\ \mathit{fail})\\ \{*r = gcd(x^{\mathsf{c}}\, y)\}\triangleright\end{array}$$

From (1) above, we get by arithmetic reasoning and (wk)

$$\{\}\ r \leftarrow \mathit{new}\ x; s \leftarrow \mathit{new}\ y\ \{\neg r = s \wedge gcd(*r^{\mathsf{c}} * s) = gcd(x^{\mathsf{c}}\, y)\}^{\mathsf{c}} \quad (5)$$

and the result now follows by applying (seq) to (4) and (5).

6 Conclusion and Future Work

We have generalized Moggi's Hoare calculus for the state monad to a *monad-independent* Hoare calculus. To this end, we have used an extension of the wide-spectrum language HasCasl with Haskell-style imperative programming via monads, which in terms of additional language features required essentially no more than a rather straightforward encorporation of constructor classes.

We have illustrated this approach by several example monads, some of which we have described *axiomatically*, rather than via an implementation as in Haskell. Specific monads come with specific extensions to the generic Hoare calculus; it is even possible to axiomatize a monad by means of Hoare triples. Further work will include the production of a library of such monad specifications, thus providing a broad basis for formal reasoning about imperative functional programs. Moreover, the examples suggest that general results about monad combination [9] bear some relation to the combination of monad-specific Hoare calculi; ways in which axiomatizations of more complex monads can be compositionally obtained those of simpler ones are the subject of further investigation. Another interesting topic is the relation to the monad independent aspects of the testing tool QuickCheck [1].

The traditional Hoare calculus can be embedded into propositional dynamic logic [6], which allows for rather more flexibility. However, this is expected to work with monads only in special cases, since unlike as with Moggi's evaluation logic, the computation needs to be split (i.e. the 'state' needs to be duplicated) for evaluations of compound formulas, e.g. conjunctions such as $[p]\,\square \wedge [q]\,\square$ where the evaluation of the second conjunct requires 'resetting the state' (here, $[p]\,\square$ reads '$\square$ holds after exection of p'). Several of the computationally relevant monads, among them the usual state monad, do admit such a state duplication; a general axiomatization of this concept is forthcoming.

Acknowledgements. This work forms part of the DFG-funded project HasCASL (KR 1191/7-1). The authors wish to thank Christoph Lüth for useful comments and discussions.

References

[1] K. Claessen and J. Hughes, *Testing monadic code with QuickCheck*, Haskell Workshop, ACM, 2002, pp. 65–77.
[2] CoFI, *The Common Framework Initiative for algebraic specification and development, electronic archives*, `http://www.brics.dk/Projects/CoFI`.
[3] CoFI Language Design Task Group, CASL – *The CoFI Algebraic Specification Language – Summary, version 1.0*, Documents/CASL/Summary, in [2], July 1999.
[4] E. W. Dijkstra, *A discipline of programming*, Prentice Hall, 1976.
[5] J.-C. Filliâtre, *Proof of imperative programs in type theory*, Types for Proofs and Programs, LNCS, vol. 1657, Springer, 1999, pp. 78–92.
[6] R. Goldblatt, *Logics of time and computation*, CSLI, 1992.
[7] *The Haskell mailing list*, `http://www.haskell.org/mailinglist.html`, May 2002.
[8] S. P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler, *Haskell 98: A non-strict, purely functional language*, (1999), http://www.haskell.org/onlinereport.
[9] C. Lüth and N. Ghani, *Monads and modularity*, Frontiers of Combining Systems, LNAI, vol. 2309, Springer, 2002, pp. 18–32.

[10] S. Mac Lane, *Categories for the working mathematician*, Springer, 1997.
[11] E. Moggi, *Notions of computation and monads*, Inform. and Comput. **93** (1991), 55–92.
[12] P. D. Mosses, Casl*: A guided tour of its design*, Workshop on Abstract Datatypes, LNCS, vol. 1589, Springer, 1999, pp. 216–240.
[13] F. Regensburger, *HOLCF: Higher order logic of computable functions*, Theorem Proving in Higher Order Logics, LNCS, vol. 971, 1995, pp. 293–307.
[14] L. Schröder and T. Mossakowski, HasCasl*: Towards integrated specification and development of Haskell programs*, Algebraic Methodology and Software Technology, LNCS, vol. 2422, Springer, 2002, pp. 99–116.
[15] Philip Wadler, *How to declare an imperative*, ACM Computing Surveys **29** (1997), 240–263.

Visual Specifications of Policies and Their Verification*

Manuel Koch[1] and Francesco Parisi-Presicce[2,3]

[1] Freie Universität Berlin, Berlin (DE)
mkoch@inf.fu-berlin.de
[2] Univ. di Roma La Sapienza, Rome (IT)
parisi@dsi.uniroma1.it
[3] George Mason University, Fairfax VA (USA)
fparisi@ise.gmu.edu

Abstract. The specification of policies is a crucial aspect in the development of complex systems, since policies control the system's behavior. In order to predict a possibly incorrect behavior of the system, it is necessary to have a precise specification of the policy, better if described in an intuitive formalism. We propose policy specifications in three modeling notations, viz. UML, Alloy and Graph Transformations, and compare them from the viewpoint of readability, verifiability as well as tool support. We use a role-based access control policy as example policy.

1 Introduction

Policies are used to control the behavior of complex systems by a set of policy rules. Policy specifications are developed by system designers and deployed by administrators. To prevent an incorrect behavior of the system due to design or administration errors, a policy specification should be readable and understandable by both the designer during system development and the administrator who must deploy the policy. Visual modeling notations are designed to enhance the understandability of system aspects. The UML [13] as the de-facto standard modeling language in industry is a widely known member. Beside a readable policy specification, a formal specification of the policy is necessary to reason about the behavior of a policy and is a prerequisite for an effective analysis of conflicts within and between policies and their resolution to achieve consistency.

The aim of our investigation is to provide support for the design of a policy by proposing an intuitive visual representation along with a formal theory to develop and modify a policy in a systematic way. We investigate in this article to what extent the modeling notations UML, Graph Transformations [17] and Alloy [20] are suited for the specification and verification of policies. UML, as the standard modeling language in industry accompanied with several tools, is considered by showing how a policy can be specified in UML in such a way that we can make use of existing UML tools. We propose a formal graph-based

* Partially supported by the EC under Research and Training Network SeGraVis.

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 278293, 2003.

semantics for the UML policy specification to make possible verification. Alloy, a lightweight object modeling notation, can express a useful range of structural properties in object models which can be automatically analyzed. Alloy provides a visual notation for parts of an object model. Graph Transformations (GT) are a graphical and formal specification technique, which incorporates both an intuitive visual notation and a formal background. Several works have reported on the use of GT for the specification of access control policies [8,9].

The paper is organized as follows. Section 2 investigates the necessary components of a policy specification and introduces the role-based access control policy example [18] used throughout the paper. Section 3 presents our proposal for a policy specification in UML, Sect. 4 the GT specification and Sect. 5 the Alloy specification. In Sect. 6, we investigate to what extent the specifications can be used to reason about the consistency of a policy specification and in Sect. 7 we compare the policy specifications from the point of view of readability, verifiability and tool support. Section 8 contains concluding remarks and future work.

2 Policies

Policies are employed to control the behavior of complex systems by using a set of policy rules that define the choices in the individual and collective behavior of the entities in the system. Beside the policy rules, declarative policy constraints may provide additional useful information on the intended behavior so that a policy specification contains also declarative information ("invariants") on what a system state must contain (positive) and what it cannot contain (negative). The declarative constraints provide useful information during the development of a policy through successive refinement steps, or when trying to predict the behavior of a policy. Therefore, a policy specification consists of 1) the type information of the system entities to which the policy applies, 2) a set of policy rules which build the accepted system states, and 3) a set of positive and negative declarative policy constraints for the wanted and unwanted substates.

2.1 Example: Role-Based Access Control

Role-based access control (RBAC) [18] reduces the complexity and cost of security administration in large systems because roles serve as a link between permissions (e.g., read or write) for objects (e.g., a file, a printer) and users. A user can access an object if (s)he plays a role which has the required permissions. The work on RBAC systems has been subject to a NIST standard proposal [19] which characterizes a family of RBAC models. The RBAC model in this article includes a role hierarchy, defining a sub-role (resp. super-role) relation between roles, whereby roles acquire the permissions of their sub-roles, and sub-roles acquire the user membership of their super-roles. The role hierarchy can be an arbitrary partial order and is assumed to be fixed. The RBAC policy rules are:

Rule 1: A user can be assigned to a role to become a member of the role. The user is authorized for a role, if this role is a sub-role of the role to which the user is assigned.

Rule 2: A user u can be revoked from a role r. We consider here only weak revocation, which revokes u only from this role r. Weak revocation has the property that a user u after revocation from r does not have to loose the permissions of r if u is assigned to a super-role of r, since super-roles inherit the permissions of their sub-roles. In strong revocation, the user u would be additionally revoked from all the super-roles of r.

Rule 3: A user can establish a session.

Rule 4: A user can close a session.

Rule 5: During a session of a user, the user can activate a subset of roles for which (s)he is authorized.

Rule 6: The user of a session can deactivate roles from the session.

The role-permission assignment [3] is assumed to be fixed. The following constraints are examples of additional restrictions of the RBAC model.

Separation of Duty (sod): This relation on roles places constraints on the assignments of users to roles. Membership in one role prevents the user from being a member of one or more other roles, depending on the sod-rules.

Cardinalities: This kind of constraint restricts the number of user-role assignments, user-session assignments etc. For example, a session must belong to a unique user.

The next sections present how the system entity types, the policy rules and the policy constraints are specified in UML, in Alloy and in Graph Transformations, exemplified with the RBAC model.

3 UML Specification

The type information of the system entities is specified in a UML class diagram. The class diagram for the RBAC example consists of the class *Role* for RBAC roles, *User* for users and *Session* for sessions (see Fig. 1a). An association shows, which class instances can be related. The label represents the intended meaning assigned to the association, its direction the direction to read the label (e.g., a *session belongs* to a *user*). The association *sod* on class *Role* specifies the separation of duty relation on roles, the association *is* represents the role hierarchy. A role r can be a super- or sub-role with respect to a role r'. Each role has a unique super-role, but a role can have zero or more sub-roles. If there is no multiplicity attached to an association, we assume the multiplicity $0..*$. The association *is_in* models the user-role assignment, the association *has_activated* models the roles that are activated in a session. The association *belongs* specifies the assignment of a session to a user.

The policy rules are specified in object diagrams using the constraints *new, all, destroy* and *destroy all* based on the constraints *new* and *destroyed* used in collaboration diagrams [13]. The intended meaning of the constraint *all* at an object in a policy rule is that all objects of this type must be considered which

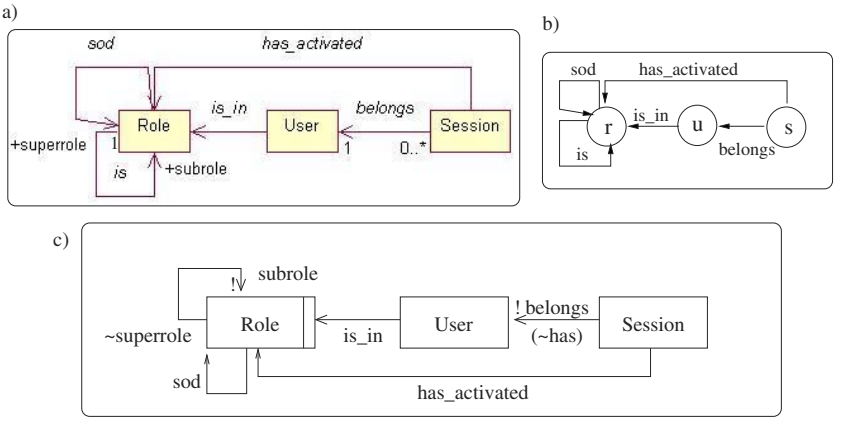


Fig. 1. Visual modeling of the type information in UML a), in GT b) and in Alloy c)

match to the object structure specified in the object diagram. An object or link with *new* is created by the policy rule, an object or link with *destroy* is removed from the system. The constraint *destroy all* at an object/link specifies that the policy rule deletes all objects/links that match the object structure specified in the object diagram. The match must be complete in the sense, that all mapped objects can be completed by links to achieve the object diagram in the rule.

Figure 2 shows the UML object diagrams for the policy rules of the RBAC model. Object diagram 1) consists of one object of type *User* carrying a constraint *new*. This specifies the creation of a user. Diagram 2) shows a user object connected to a session object. The constraint *destroy* at the user object specifies that the policy rule deletes the user, the constraints *destroy all* at the session object and the link specify that all sessions of the user are deleted, as well. Note, that sessions which are not connected to the deleted user object remain unchanged. Diagram 3) specifies the creation of a new session object connected to a user object. The user object already exists, only the session object and the link to the user are added by the policy rule. Diagram 4) specifies the deletion of a session and the connection to the user, diagram 5) the assignment of an existing user to an existing role by adding a new link. The revocation of a user from a role is specified in diagram 6). If a *user* is revoked from *role* (specified by destroying the link between the objects *user* and *role*), *user* may loose the authorization for all sub-roles of *role*. Therefore, all sub-roles of *role* must be deactivated from the sessions of the user, as well. The $*$ attached to the link between the roles *role* and *role'* models a path through the role hierarchy from *role* to *role'*. Since UML has no primitive operation for a transitive closure, transitivity must be specified in OCL. Therefore, we introduce the operation `closure()` on roles:

```
class Role
  closure(): Set(Role) =
    subrole.closure()->asCollection->including(self)
```

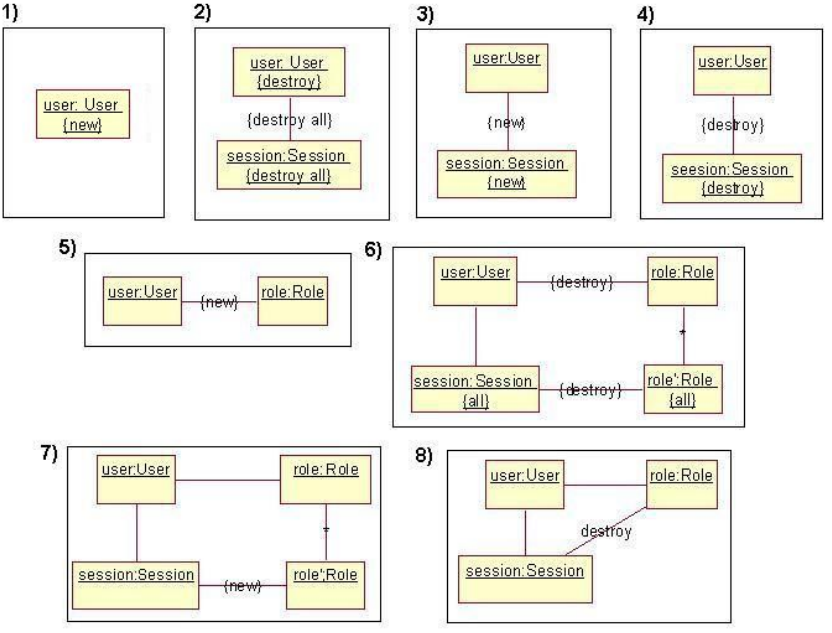


Fig. 2. UML object diagrams for policy rules

The *-link between objects *role* and *role'* in diagram 6) is a graphical notation for *role'* in *role.closure()*. Since we assume a non-cyclic role hierarchy, the operation `closure()` always terminates. Diagram 7) specifies the activation of a role in a user session by creating a new link between the session object and a role for which the user is authorized. A user is authorized for a role $role'$ if there is path starting from a role to which the user is assigned ending in $role'$ (specified by the *-link). Diagram 8) specifies the deactivation of a role from a session by destroying the link between the session object and the role object.

To specify policy constraints, we have chosen OCL [21]. The OCL constraint below specifies the separation of duty (sod) constraint.

```
context User inv
   self.is_in->forAll( r1,r2 | r1.sod->excludes(r2) )
```

The RBAC cardinality constraint, which requires a unique user for each session, is already specified by the multiplicity 1 at the association *belongs* in the class diagram in Fig. 1 a).

4 Graph Transformations Specification

Graph Transformations (GT) [17] provide an intuitive presentation of graph-based structures and their rule-based modification. The types of the system

entities are specified in a *type graph*. A type graph is a graph consisting of a set of nodes and a set of directed edges which specify the types of nodes and edges which may be used in the instance graphs modeling system states. A graph G is an instance graph of a type graph if one can find for each node and edge in G the corresponding node and edge type in the type graph. We take a GT approach in which edges are relations, i.e., there is at most one edge of the same type between two nodes, but there can be several edges of different type between the same nodes. Fig. 1b shows the type graph for the RBAC policy model. It has the types u for user, r for roles and s for sessions. The meaning of the edges corresponds to the meaning in the UML class diagram in Fig. 1a. In contrast to class diagrams, however, it is not possible to specify multiplicities in a type graph. An edge in a type graph always has the multiplicity $*$. Furthermore, edges in the type graph do not carry role names as associations in UML class diagrams.

The policy rules are specified by *graph rules*. Formally, a graph rule is given by a *graph morphism* $r : L \to R$ which consists of two partial injective mappings: one between the set of nodes and one between the set of edges of L and R, so that 1) whenever the mapping for edges is defined for an edge e pointing from node s to node t, the mapping for s and t is defined and the edge $r(e)$ in R points from $r(s)$ to $r(t)$ and 2) node/edges are mapped only to nodes/edges of the same type. We call the graph morphism *total* if the mappings between the node and edge sets are total. The graph L of a graph rule $r : L \to R$, *left-hand side* (LHS), describes the elements a graph must contain for r to be applicable. The morphism r is undefined on nodes/edges that are intended to be deleted, defined on nodes/edges that are intended to be preserved. Nodes and edges of R, *right-hand side* (RHS), without a pre-image are newly created. Note that the actual deletions/additions are performed on the graphs to which the rule is applied. The application of a rule to a graph G requires an occurrence of the LHS L in G (formally defined by the existence of a total graph morphism $m : L \to G$, called match). The application itself consists of two steps. First, delete all objects in G that have a pre-image in $L \setminus dom(r)$. Second, add all graph objects of $R \setminus r(L)$ to G connected to the nodes $m(dom(r))$.

The graph rules for the RBAC model are given in Fig. 3. The LHS of the graph rule *add user* is empty, its RHS contains one user node. This rule specifies the creation of a new user. The rule *remove user* deletes a user and all its connected sessions. The double circle around the session node specifies that all sessions connected to the user are deleted. The rule *new session* creates a new session for a user by inserting a new session node connected to the user node. The LHS of the graph rule *remove session* consists of the session node connected to the user node, the RHS consists only of the user node. This graph rule specifies the deletion of the session. The rule *add to role* assigns a user to a role. The rule *remove from role* specifies the revocation of a user from a role. The edge between the user and the role is deleted as well as all edges between sessions and roles for which the user is authorized by the revoked role. The * specifies a path between the roles. The rule *activate role* inserts an edge between a session of a user and a role for which the user is authorized. The graph rule *deactivate role*

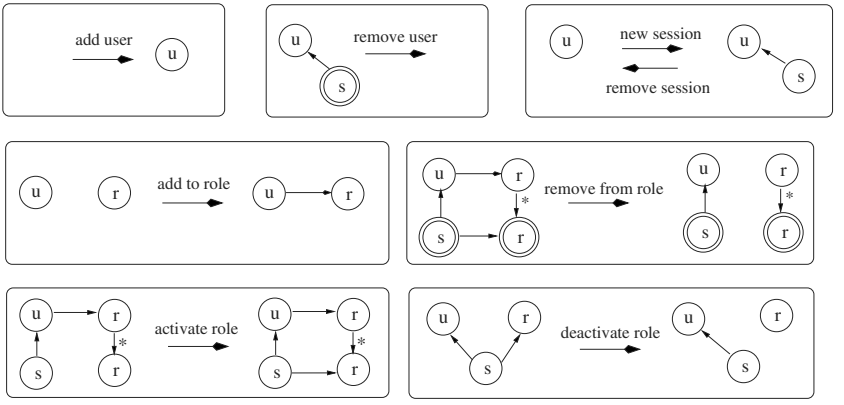


Fig. 3. Graph transformation rules for policy rules

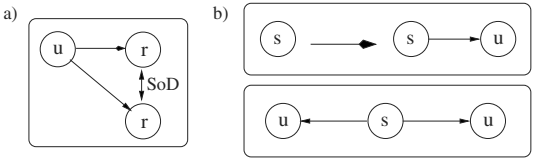


Fig. 4. Graphical constraints for policy constraints

deactivates a role from a session of a user by deleting the edge between session and role node.

Policy constraints are specified by *positive* and *negative graphical constraints* [10]. A positive graphical constraint (PGC) is a total graph morphism $c : X \to Y$ and a graph G satisfies c if for all total morphisms $p : X \to G$ there is a total morphism $q : Y \to G$ so that $q \circ c = p$. A negative graphical constraint (NGC) is a graph C and a graph G satisfies C if there does not exist a total morphism $p : C \to G$. The NGC in Fig. 4 a) specifies the separation of duty constraint. The NGC shows the forbidden substate in which a user is assigned to two roles in sod-relation. The two graphical constraints in b) specify that each session belongs to a unique user. The PGC requires for each session at least one connected user, the NGC forbids two (or more) users connected to one session.

5 Alloy Specification

Alloy[6] is a language for describing structural properties of object models and provides constraints and operations to describe how object structures may change. It is a state-based language with a textual syntax together with a graphical sublanguage. Alloy allows the designer to automatically analyze specifications [7]. The RBAC model of this paper is a submodel of the model used in [20].

The types for the system entities are specified in the Alloy diagram in Fig. 1c. Each box in the diagram represents a set of objects. We have the sets *Role*, *User* and *Session*. A vertical stripe down the right-hand side of the box specifies a fixed set, i.e., the number of elements in this set is fixed throughout the lifetime of the system. In the RBAC example, the roles are fixed, but the sets of users and sessions are not fixed. An edge represents a relation: for example, the relation *sub-role* maps a role to all its sub-roles, the relation *super-role* is defined to be its inverse (specified by the tilde). The relation *super-role* maps each role to its unique super-role. The exclamation mark attached to the end of a relation means exactly one. Another example is the relation *belongs* which maps each session to a unique user and its inverse *has* maps each user to his/her activated sessions.

Policy rules are specified textually by Alloy operations (keyword `op`). The primed values in operations indicate the post state of the variables. For sets, there are the usual set-theoretic operations for union (+) and difference (-), the operator `s in t` checks if the set `s` is a subset of `t`. For a relation `r`, the operation $\sim$`r` is the inverse of `r` and $*$`r` is the reflexive transitive closure of `r`. The '.' operator is used to specify a navigation expression, i.e., `s.r` denotes the set of objects that the set `s` maps to in the relation `r`. For example, `user.is_in` specifies the set of roles the `user` plays in a state.

```
op NewUser(user:User□) { User□ = User + user }

op RemoveUser(user:User□) {
  Session□ = Session - user.□has
  User□ = User - user
}
op NewSession(user:User, session:Session□) {
  Session□ = Session + session
  session.belongs□ = session.belongs + user
}
op RemoveSession(user:User, session:Session□) {
  user.□has□ =  user.□has - session
  Session□ = Session - session
}
op AddToRole( user:User, role:Role ) {
  user.is_in□ = user.is_in + role
}
op RemoveFromRole(user:User, role:Role) {
   user.is_in□ = user.is_in - role
   user.□has.is_activated□=user.□has.is_activated-role.*subrole
}
op ActivateRole(user:User,session:Session,role:Role,role2:Role){
   role in user.is_in
   role2 in role.*subrole
   session.is_activated□ = session.is_activated + role2
}
```

```
op DeactivateRole(user:User, session:Session, role:Role) {
   session.is_activated' = session.is_activated - role
}
```

Policy constraints are described by Alloy assertions (keyword `assert`), which are questions of the kind "Is it true that ...?". The Alloy analyzer tries to answer this question by finding a counterexample (more in Sect. 6). The assertion for the sod-constraint states that when a user u is assigned to roles $r1$ and $r2$ then $r1$ and $r2$ are not in sod-relation (`->` specifies implication, `!` negation).

```
assert SeperationOfDuty{ all r1,r2: Role, u:User |
                         (r1 + r2) in u.is_in -> r1 != r2.sod }
```

The cardinality constraint of a unique user for each session is visually specified by the exclamation mark attached to the relation *belongs* in the Alloy diagram.

6 Verification

A crucial property of a policy specification is that it specifies a coherent policy in the sense that the policy rules and the policy constraints are not contradictory. Therefore, we define a policy specification to be *coherent* if all system states built by the policy rules satisfy the constraints. In the context of UML, a system state is an object model and a policy constraint an OCL constraint. Satisfaction deals with the satisfaction of the OCL constraint in the object model. In the context of Alloy, a system state is an instance of the Alloy model with concrete instance sets and relations. A policy constraint is an Alloy assertion and satisfaction deals with the satisfaction of the assertion in the instance model. In the context of GT, a system state is a graph and a policy constraint is a graphical constraint. Satisfaction deals with satisfaction of the graphical constraint by the state graph.

The RBAC policy specifications in the previous sections are not coherent, since the sod-constraint is not satisfied by all system states built by the policy rules. The policy rule for the assignment of a user to a role (diagram 5 in Fig. 2 in the case of UML, graph rule *add to role* in Fig. 3 in the case of GT, operation `AddToRole` in the case of Alloy) does not consider the sod-relation. The rule can assign a user to any role and may construct a system state which does not satisfy the sod-constraint. This section investigates the problem of detecting incoherence in a policy specification and, if so, of resolving it.

6.1 Alloy

Alloy is supported by an analyzer [7] to detect constraint violations. The analyzer checks Alloy assertions by trying to find a counterexample, i.e., to find an instance model of the Alloy specification in which the assertion is not satisfied. Using the Alloy analyzer to check the sod assertion of the policy specification in Sect. 5, the analyzer gives the counterexample in Fig. 5. The $User1$ is assigned to the roles $Role2$ and $Role3$ which are in sod-relation.

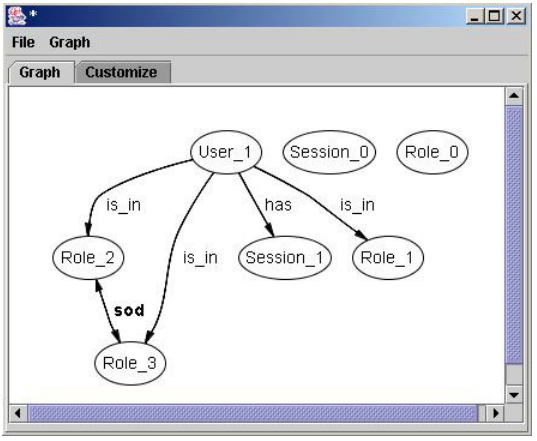


Fig. 5. Counterexample found by the Alloy analyzer

Based on the counterexample, the designer tries to modify the Alloy specification to resolve the conflict. In our example, the invariant `assign` is added to the Alloy specification. It specifies that roles $r1$ and $r2$ both assigned to a user u must not be in sod-relation. Applying the Alloy analyzer to the extended Alloy specification results in no solution, i.e., no counterexample could be found and the Alloy policy specification is coherent.

```
inv assign { all u: User, r1: Role, r2: Role |
             (r1 + r2) in u.is_in -> r1 !in r2.sod }
```

6.2 Graph Transformations

Graph Transformations provide a constructive approach to the detection of constraint violation [4,8]. The algorithm receives as input a graph rule and a graphical constraint and decides whether the graph rule may construct a graph which does not satisfy the graphical constraint. When a conflict is detected, the algorithm modifies the graph rule by adding a negative application condition (NAC) so that the modified graph rule is applicable only when it is guaranteed to produce a new graph that satisfies the graphical constraint. A NAC for a graph rule $r : L \to R$ is a total graph morphism $n : L \to N$ and r with NAC n can be applied to a graph G if there is a match $m : L \to G$, but there is no total morphism $q : N \to G$ so that $q \circ n = m$.

Applying the algorithm to the graph rule *add to role* in Fig. 3 and the graphical constraint for sod in Fig. 4, the algorithm detects that the rule may violate the constraint. Therefore, the algorithm adds a NAC to the rule (Fig 6 a)). In the representation of a graph rule with NAC, the nodes and edges of L are drawn by solid lines, the part $N \setminus n(L)$ by dotted lines. The NAC for rule *add to role* is given by the dotted role node r', the dotted edge between the user u and role

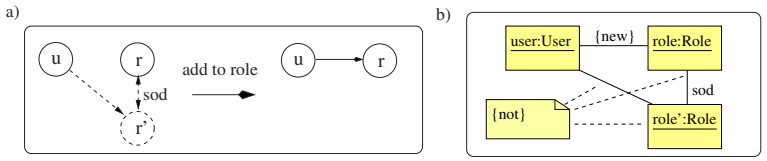


Fig. 6. Consistent graph rule *add to role* and its translation into a UML policy rule b)

r' as well as the dotted sod-edge. The NAC forbids the assignment of a user to a role r if the user is assigned to a role r' which is in sod-relation with r.

6.3 UML A Graph-Based Formal Semantics

There is some tool support for OCL constraints ranging from syntactical analysis, type checking, dynamic invariant validation, dynamic pre-/post-condition validation or test automatization [16,5]. Since our aim is to check the coherence of a policy specification in UML, we focus on the validation of invariants supported, for example, by the USE tool [15,14]. The USE tool allows the designer to validate OCL constraints against snapshots of the system. To check the coherence of a policy specification, however, we have to consider all system states that can be built by the policy rules. Since it is not possible to generate all these states by the tool, it is not suitable to check the coherence of a policy.

Therefore, we propose a translation from the UML policy specification into a GT specification, based on the results in [11]. The translation of a class diagram into a type graph is essentially straightforward: each class x becomes a node x, each directed association a becomes an edge a (for undirected associations a we get two edges pointing in opposite direction). The multiplicities in a class diagram are converted into graphical constraints. A multiplicity range n⋙m ($n < m$) for an association is translated into a positive graphical constraint (PGC) for the lower bound n and a negative graphical constraint (NGC) for the upper bound m. The graph X of the PGC $c : X \to Y$ for the lower bound n contains one object node of the association's source class. The graph Y contains the same node and n object nodes for the association's target class. The morphism c maps the object node in X to its counterpart in Y. The graph C of the NGC for the upper bound m contains one object node of the association's source class and $m + 1$ objects of the association's target class. For example, the multiplicity 1 at association *belongs* is translated into the graphical constraints in Fig. 4 b).

The UML object diagrams for policy rules are translated into graph rules $r : L \to R$, where L is the graph given by all objects/links without constraint *new* and R is the graph given by all objects without *destroy* or *destroy all*. Objects with constraint *all* or *destroy all* become a node with a double circle. Associations with attached $*$ are translated to edges carrying a $*$. The rule morphism r is defined for all objects/links without *new* and maps the objects/links to their counterparts in R. The graph rules in Fig. 3 are the result of the translation of

the UML policy rules in Fig. 2. The Fujaba system [12] implements the idea to use UML collaboration diagrams as a front-end notation for GT rules.

Whereas the translation of the class diagram and the policy rules can be done automatically due to a similar visual presentation, the translation of OCL constraints in graphical constraints must be done by the designer. In [1] an automatic translation of OCL constraints into graphical constraints is shown. But only simple OCL constraints can be translated. Another possibility is a direct visual specification of UML policy constraints as proposed in [11]. When the OCL constraints are translated into graphical constraints, the coherence of the policy specification can be checked by the algorithm in 6.2. To translate a graph rule with NAC into an object diagram for the UML policy rule, we attach a note with constraint *not* to all objects/links coming from the NAC. Fig. 6b shows the translated UML object diagram from the graph rule in a).

7 Comparison

Our aim is a policy specification framework that provides an intuitive visual notation usable by policy designers, deployers and administrators along with a formal theory to reason about the coherence of a policy specification. The framework would benefit from a tool support for both specification and verification. We compare to what extent the presented approaches bring forth this aim. Table 1 shows a summary of the comparison. Column *visual* states which parts of a policy can be specified visually, column *verification* states if there are concepts to check constraints or to resolve incoherence, column *tool* states the tool support for the specification and verification of policies.

Table 1.

	visual	verification	tool
UML	types, policy rules	no, but translation into GT possible	specification
Alloy	types	constraint checker	specification, verification
Graph Transformations	types, policy rules, policy constraints	constraint checker, conflict resolving	specification

7.1 Visual Specification

The visual specification of the type information of a policy is quite similar in all three approaches. The diagrams in Fig. 1 describe almost the same graph and differ only in small parts. For example, both the UML class diagram and the Alloy diagram provide the specification of multiplicities for associations/relations,

whereas the GT type graph specifies only the existence or non-existence of connections. Multiplicities in GT must be specified separately in graphical constraints, which is less intuitive than the direct presentation in the diagram.

The difference between the approaches becomes more significant when policy rules and policy constraints are specified. Graph Transformations rules provide an intuitive graphical notation to specify both the triggers of a policy rule, i.e., when to apply a policy rule, and the policy rules' effects in one compact representation. UML provides several possibilities which could be used to describe the policy rules. As an alternative to our proposal of object diagrams, sequence diagrams could be used, in which messages specify both the triggers of a policy rule and its effects. Messages may cause a change of the object structure. Since the order of the messages for policy rules is less interesting (the trigger is the first message, the message order for the policy rule effects is generally not important) than the change of the object structure, collaboration diagrams appear to be more suitable. In fact, the object diagrams chosen in our approach are the context of collaboration diagrams without any interaction. Our approach can therefore be easily extended, if interaction must be considered in policy rules, as well. The expressiveness of Alloy operations allows the designer to specify policy rules, but there is no graphical representation for Alloy operations (yet). It is possible, however, to provide also a visual notation to Alloy similar to the one proposed for UML.

Visual constraint specification is only possible in GT. Neither the OCL constraints nor the Alloy assertions/invariants can be presented visually. As mentioned above, simple OCL constraints, however, could be represented also graphically [1].

7.2 Verification

Given a policy specification, we want to determine if the policy rules may violate the policy constraints. The Alloy analyzer checks the constraints by looking for a counterexample, i.e., an instance model produced by the policy rules which does not satisfy the constraint. If a counterexample is found, however, the Alloy analyzer does not say which operation (i.e., policy rule) causes the inconsistency. The designer has to interpret the counterexample and change the Alloy specification for a resolution.

In GT, constraints and rules are checked pairwise to detect inconsistencies. When a conflict is detected, the algorithm gives the designer the graph rule and the graphical constraint which cause the conflict. Unlike with Alloy, the conflict is automatically solved by modifying the graph rule and maintaining the constraint.

OCL constraints can be checked only w.r.t. snapshots of the system, but there are no concepts in UML to check the coherence of a policy, i.e., whether the policy rules may build a system state in which an OCL constraint is not satisfied. As shown in 6.3, the UML specification can be translated into a GT policy specification to use GT verification concepts.

7.3 Tool Support

Tool support for the specification of a policy is partly available for all three notations. UML is accompanied by several CASE tools and our approach to the UML policy specification can be written in existing UML tools, since only standard UML extension mechanisms are used. The Alloy policy specification can be written in the Alloy analyzer tool [7]. The specification for the analyzer, however, is completely textual and does not support the graphical notation presented in Sect. 5. For the specification of the GT policy specification, general GT tools can be used [2].

Tool support for the verification of the coherence of a policy is only available in the Alloy analyzer. There is no tool support for checking the coherence of a policy specification either in UML or in GT, since the algorithm for detecting and resolving constraint violations in GT is not yet implemented.

8 Concluding Remarks

To reduce errors in the behavior of complex systems due to faulty policy specifications or due to administration errors of correct, but incomprehensible, policy specifications, we have presented policy specifications in UML, Alloy and GT, providing a concise notation as visual as possible and a formal semantics to reason about policy coherence. We have proposed a way to express the policy components in UML so that, on the one hand, UML tools can be used and, on the other hand, a formal semantics based on GT can help to reason about the coherence of a policy. Future work could investigate the translation of a UML policy into an Alloy policy to use the Alloy analyzer in the UML context.

None of the notations, however, reaches our aim completely. In Alloy and UML, not all policy components, especially constraints, can be expressed visually. This is possible in GT, but the tool support to check the policy is missing, even if the theoretical results do exist. The tool support for the verification is an advantage of Alloy. Future work will deal with a tool which implements the GT checking algorithm and the automatic translation of an XMI representation of a UML policy specification into a GT representation. This allows the designer to import UML policies and to check their coherence by means of the GT checking algorithm.

To get practical results about the actual understandability, the proposed notations must be further evaluated (preferably) by user tests. Moreover, these tests would help to refine the proposed notations to missing concepts needed in policy specifications or to adapt the notations to special user requirements. For example, the UML notation of this paper is one possibility to specify a policy in UML to which a GT semantics can be given. But there may be other UML representations (e.g., using statecharts/sequence diagrams) that are more convenient for software engineers and clients.

References

1. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In *Proc. UML2000*, number 1939 in LNCS, 2000.
2. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. II: Applications, Languages, and Tools*. World Scientific, 1999.
3. P.A. Epstein. Engineering of Role/Permission Assignments. *PhD Thesis, George Mason University*, 2002.
4. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In *Proc. SEGRAGRA'95 Graph Rewriting and Computation*, number 2. Electronic Notes of TCS, 1995. `http://www.elsevier.nl/locate/entcs/volume2.html`.
5. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. In *Proc. of UML2000*, volume 1939 of *LNCS*, pages 278–293. Springer, 2000.
6. D. Jackson. Alloy: A Lightwight Object Modelling Notation. Technical Report 797, MIT Laboratory for Computer Science, 2001.
7. D. Jackson, I. Schlechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, 2000.
8. M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Graph Based Formalism for RBAC. *ACM Transactions on Information and System Security (TISSEC)*, 5(3):332–365, August 2002.
9. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Lect. Notes in Comp. Sci. Springer, March 2001.
10. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Conflict Detection and Resolution in Access Control Specifications. In M.Nielsen and U.Engberg, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2002)*, Lect. Notes in Comp. Sci., pages 223–237. Springer, 2002.
11. M. Koch and F. Parisi-Presicce. Access Control Policy Specification in UML. In *Proc. of UML2002 Workshop on Critical Systems Development with UML*, number TUM-I0208, pages 63–78. Technical University of Munich, September 2002.
12. U. Nickel, J. Niere, and A. Zündorf. Tool demonstration: The FUJABA environment. *Proc. of the 22nd Int. Conf. on Software Engineering*, 2000.
13. OMG. OMG Unified Modeling Language Specification, V.1.4, 2001.
14. M. Richters. The USE tool: A UML-based specification environment, 2001. `http://www.db.informatik.uni-bremen.de/projects/USE`.
15. M. Richters and M. Gogolla. Validating UML Models and OCL Constraints. In *Proc. UML2000*, 2000.
16. M. Richters and M. Gogolla. OCL – Syntax, Semantics and Tools. In *Proc. Advances in Object Modelling with OCL*, LNCS, pages 38–63. Springer, 2001.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*. World Scientific, 1997.
18. R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. In *1st ACM Workshop on Role-based access control*, 1996.
19. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST Model for Role-Based Access Control: Towards A Unified Standard. In *Proc. of the 5th ACM Workshop on Role-Based Access Control*. ACM, July 2000.

20. A. Schaad and J.D. Moffett. A Lightweight Approach to Specification and Analysis of Role-based Access Control Extensions. In *Proc. 7th ACM Symposium on Access Control Models and Technologies*. ACM Press, 2002.
21. J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison Wesley, 1999.

Automatic Model Driven Animation of SCR Specifications

Angelo Gargantini[1] and Elvinia Riccobene[2]

[1] C.E.A.– Università di Catania
a.gargantini@unict.it
[2] Dipartimento di Matematica e Informatica – Università di Catania
riccobene@dmi.unict.it

Abstract. This paper introduces *automatic model driven animation*, a novel approach to validate requirements specifications. This approach, here applied to SCR specifications, is based on graphical animation. Automatic model driven animation consists in automatically deriving scenarios from requirements specifications; these scenarios are used to animate critical system behaviors through a graphical interface. Animation is useful at the very early stages of systems development to better understand models and requirements, to gain confidence that specifications capture informal requirements, and to detect faults. We introduce a technique, exploiting model checkers, to automatically generate animation sequences starting from requirements specifications, and we present a prototype tool for the generation and animation of scenarios.

1 Introduction

Many methods, techniques, and tools are continuously proposed to analyze, validate, and verify formal specifications. Several methods deal especially with validation of formal specifications. The validation activity can be summarized by the well known question "Are we building the right system?" and it consists in checking whether the system, as specified, meets the user needs. Techniques for validation include scenarios generation, development of prototypes, simulation, and also testing [17,7]. These techniques can help designers and customers to better understand models and requirements, to gain confidence that specifications capture informal requirements, and to detect faults in specifications as early as possible with limited effort. Validation should precede the application of more expensive and accurate methods, like formal requirements analysis and formal verification of properties [8], that should be applied only when designers have enough confidence that specifications are correct. To be effective, the validation activity should be supported by tools easy to use and requiring minimum user effort.

This paper introduces *automatic model driven animation* as a novel approach to validate requirements specifications. This approach, here applied to SCR (Software Cost Reduction) specifications [13], is useful at the very early stages of systems development, and it is based on *graphical animation* (hereinafter briefly called *animation*) [10,16,19]. Animation basically consists of simulation, providing the user with a graphical interface suitable to show the state of the system by means of icons, buttons, panels, slides, and so on. An animator might be a prototype ad hoc developed or a simulator endowed with

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 294–309, 2003.

a complex graphical interface. A prototype tool for animation of SCR specifications, is described in this paper.

Animation offers several advantages. Mainly, through animation designers better understand the requirements specification and can find failures and faults. For customers, looking at the real behavior shown by the animator is better than reading mathematical or logical formulas modeling the system behavior (normally customers do not like mathematical formalisms). Customers can ignore in which notation the specification has been written and they do not need to learn new (formal) languages.

Animation, especially as proposed in this paper, does not require skills, ingenuity and expertise as those required by *heavy* formal methodologies like theorem proving. However, animation cannot prove that a specification is correct, but it can only uncover faults [21]. For this reason, it is very similar to testing.

The main problem of animation is the selection of system behaviors to animate. We distinguish three main ways for selecting scenarios:

1. *user driven animation*: users (customers or designers) "play" with the animator and check whether the specification meets customers needs or not. Users select inputs, regardless the specification, by means of buttons, slides, and graphical elements; outputs are computed according to the specification (that acts as oracle) and shown by means of other graphical elements. Except for the graphical interface, this approach is very similar to the classical simulation.
2. *random animation*: inputs are randomly generated taking into account only their specified constraints. Outputs are computed according to the specification and shown through the graphical animator. In this case, the user observes the system behavior and judges its correctness. This approach is proposed in [22].
3. *model driven animation*: inputs are selected starting from requirements specifications in a systematic way, either manually or automatically. The former is similar to case 1, but the specification is used as guideline. The latter is the new approach proposed in this paper.

In any of these three animation approaches, the judgment of the specification correctness is left to the human; however, the effort required substantially differs, and model driven animation is more efficient than user driven or random animation for several reasons. By animation, designers gain confidence that the specification is correct only if the model has been extensively simulated and enough scenarios have been checked. Therefore, performing a *good* selection of critical scenarios, that can uncover specification faults, is crucial. Since random animation produces a huge amount of scenarios, the careful review of all the behaviors is time consuming. Furthermore, only few generated scenarios are able to expose critical faults.

In user driven, as well as in manual model driven animation, designers have the responsibility to cover all the critical behaviors. Since selection is carried out by hand, they risk to leave out some particular cases and choose only a small subset of all the specified behaviors. The manual selection of scenarios is, therefore, expensive and error-prone, especially in user driven animation, where specifications are not used as guidelines.

Automatic model driven animation, here introduced, has the advantage to automatically derive scenarios from specification, and to assure the animation of all the critical

behaviors according to the requirements. It does not require great user skill and ingenuity and is an effective approach for model validation.

Besides selection of critical scenarios, another critical issue is the actual graphical animation of the generated scenarios. This requires a tool endowed with a graphical animator panel that reproduces the look and feel of the real system under animation. This verisimilar environment is of great help to customers: they do not have to learn a new environment (like a new tool or IDE or interface), they do not watch variable values only by means of digits or strings (like in debuggers), they do not have to query system state by typing commands, and they do not need to guess the meaning of the values in terms of system behaviors. To be effective, animator panels must be constructed and modified rapidly, accurately, and cheaply. They do not have to be efficient, complete, portable, or robust and they do not have to use the same hardware, software, or implementation language as the delivered system. An animator panel should have two kinds of components: static components, as background images, logo, and frames, that do not change their aspect during simulation, and dynamic graphical components that change their aspect (color, size, or shape) according to variable changes, showing the system evolution. In this way, the user can watch the system as he/she were just in front of the real system. A real example of animator panel for SCR can be found in [11], where a complex graphical front-end of a simulator is developed to simulate an aircraft cockpit. In this work, we also present the software architecture of an animator tool endowed with an animator panel compliant with these requirements.

The paper is organized as follows. Section 2 introduces the formal method SCR and proposes a case study we will use to illustrate the novel approach and the animator prototype. Automatic animation for SCR specifications is presented in Sect. 3. We explain how to automatically compute from SCR specifications scenarios assuring the animation of all the critical behaviors. Section 3.3 introduces an interesting variant of automatic model driven animation, called *animation on demand*. In Sect. 4, a prototype tool for automatic animation of SCR specifications is presented. Related work is discussed in Sect. 5, while conclusions and future work are presented in Sect. 6.

2 Software Cost Reduction Technique

The Software Cost Reduction (SCR) [13] is a set of techniques for designing software systems developed by David Parnas and researchers from U.S. Naval Research Laboratory (NRL). SCR offers several automated techniques, supported by a tool set [11], for detecting errors in software requirements specifications, including an automated *consistency checker* to detect missing cases and other application-independent errors [14]; a *simulator* to symbolically execute the specification to ensure that it captures the users intent; and a *model checker* to detect violations of critical application properties [1].

2.1 The Formal Method

The SCR model represents the environmental quantities that the system monitors and controls as *monitored* and *controlled variables*. The environment nondeterministically produces a sequence of input events, where an *input event* signals a change in some

Table 1. Condition, event, and mode table format

Modes	**Conditions**			
m_1	$c_{1,1}$	$c_{1,2}$	...	$c_{1,p}$
...	...	...	...	...
m_n	$c_{n,1}$	$c_{n,2}$	...	$c_{n,p}$
var_{Ci}	v_1	v_2	...	v_p

Condition table

Modes	**Event**			
m_1	$e_{1,1}$	$e_{1,2}$	...	$e_{1,p}$
...	...	...	...	...
m_n	$e_{n,1}$	$e_{n,2}$	...	$e_{n,p}$
var_{Ei}	v_1	v_2	...	v_p

Event table

Old Mode	**Event**	**New mode**
m_1	$e_{1,1}$	$\bar{m}_{1,1}$
	...	...
	e_{1,p_1}	$\bar{m}_{1,p_1}$
...	...	..
m_n	$e_{n,1}$	$\bar{m}_{n,1}$
	...	...
	e_{n,p_n}	$\bar{m}_{n,p_n}$

Mode table

monitored quantity. The system, represented in the model as a state machine, begins execution in some initial state and then responds to each input event in turn by changing state and by possibly producing one or more output events, where an *output event* is a change in a controlled quantity. An assumption of the model is that at each state transition, exactly one monitored variable changes value. To concisely capture the system behavior, SCR specifications may include two types of internal auxiliary variables: *terms,* and *mode classes* whose values are modes. Mode classes and terms often capture historical information. In the SCR model, a system is represented as a 4-tuple, $(S^{\prime}\, S_0{}^{\prime}\, E^{m}{}^{\prime}\, T)$, where S is the set of states, $S_0 \subseteq S$ is the initial state set, E^m is the set of input events, and T is the transform describing the allowed state transitions [14]. Usually, the transform T is deterministic, i.e. a function that maps an input event and the current state to a new state. To construct T, SCR composes smaller functions, each derived from the two kinds of tables in SCR requirements specifications, event tables and condition tables. These tables describe the values of each *dependent variable*, that is, each controlled variable, mode class, or term. Tables have the typical format shown in Table 1. A *condition table* specifies that the value of the variable var_{Ci} is v_k if the boolean condition $c_{j,k}$ holds in mode m_j. An *event table* specifies that the variable var_{Ei} takes value v_k when event $e_{j,k}$ happens in mode m_j. *Mode tables* are a variant of event tables and specify the behavior of mode class: if mode has value m_j and event $e_{j,k}$ happens, then mode becomes $\bar{m}_{j,k}$. The SCR model requires the entries in each table to satisfy certain "consistency" and "completeness" properties. These properties guarantee that all of the tables describe total functions [14]. Tabular notation, with an intuitive semantics, facilitates the practical use in industrial applications.

In SCR, a *state* is a function that maps each variable in the specification to a value, a *condition* is a predicate defined on a system state, and an *event* is a predicate defined on a pair of system states implying that the value of at least one state variable has changed. When a variable changes value, we say that an event "occurs". The expression "`@T(c) WHEN d`" represents a *conditioned event*, which is defined by

$$@\mathtt{T}(\mathtt{c})\ \mathtt{WHEN}\ \mathtt{d} \stackrel{\text{def}}{=} \neg \mathtt{c} \wedge \mathtt{c}' \wedge \mathtt{d}$$

where the unprimed conditions c and d are evaluated in the *current* state and the primed condition c' is evaluated in the *next* state. The expression "`@T(c)`" means $\neg \mathtt{c} \wedge \mathtt{c}'$, while "`@F(c)`" means $\mathtt{c} \wedge \neg \mathtt{c}'$.

2.2 An SCR Case Study: The Safety Injection System (SIS)

Case study of this paper is the SCR specification of a system called the Safety Injection System (SIS), a simplified version of a control system for safety injection in a nuclear plant [5], which monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold. The system operator may override safety injection by turning a “Block” switch to “On" and may reset the system after blockage by setting a “Reset" switch to “On".

To specify the SIS requirements in SCR, we represent the SIS inputs with the monitored variables `WaterPres`, `Block`, and `Reset` and the single SIS output with a controlled variable `SafetyInjection`. The specification also includes two internal auxiliary variables, a mode class `Pressure`, an abstract version of `WaterPres`, and a term `Overridden` which indicates when safety injection has been overridden. The Mode Table 2 specifies the value of `Pressure` and the Event Table 3 specifies the value of `Overridden`. The Condition Table 4 specifies the behavior of `SafetyInjection`. A constant `Low` = 900 defines the threshold that determines when `WaterPres` is in an unsafe region, while a constant `Permit` = 1000 determines when the pressure is high and the system cannot be overridden.

Since tables specify the function T of transformation from the current state to the next one, they represent operational or functional requirements. Besides these requirements, SCR model introduces system properties in terms of events and conditions that must be true in every state. These properties are safety requirements and represent the declarative requirements of the system. We exploit this distinction in Sect. 3. For SIS, the property “*if pressure becomes too low, safety injection does not turn on if the system is blocked*”

Table 2. Mode table defining the Mode Class `Pressure` of SIS

Old Mode	Event	New mode
TooLow	@T(WaterPres $\geq$ Low)	Normal
Normal	@T(WaterPres $\geq$ Permit)	High
	@T(WaterPres < Low)	TooLow
High	@T(WaterPres < Permit)	Normal

Table 3. Event table for term overridden

Pressure	Events	
High	false	@F(Pressure=High)
TooLow, Normal	@T(Block=On) WHEN Reset = Off	@T(Pressure=High) OR @T(Reset = On)
Overridden	True	False

Table 4. Condition table for controlled variable SafetyInjection

Pressure	Conditions	
TooLow	Overridden	not Overridden
Normal, High	true	false
SafetyInjection	Off	On

is a safety requirement expressed in SCR as:

$$@\mathtt{T}(\mathtt{WaterPres} < \mathtt{Low})\mathtt{WHEN}\,\mathtt{Block} = \mathtt{On} \wedge \mathtt{Reset} = \mathtt{Off} \rightarrow \mathtt{SafetyInjection}' = \mathtt{Off} \tag{1}$$

i.e. if the water pressure falls below Low and Block is On and Reset is Off, then Safety Injection is Off in the next state. This property has been proved in [12].

3 Automatic Model Driven Animation

In this section, we tackle automatic model driven animation for SCR specifications. Figure 1 shows the process of generation and animation of critical scenarios. *Animation goals,* each representing a critical system behavior to animate, are systematically derived from SCR specifications in an automatic way. Scenarios achieving the animation goals are computed by exploiting the counter example generation of model checkers. Finally, animation sequences are animated by means of a simulator endowed with an animator panel. In Sect. 4, we describe the animation of critical scenarios for SIS by means of an animator prototype.

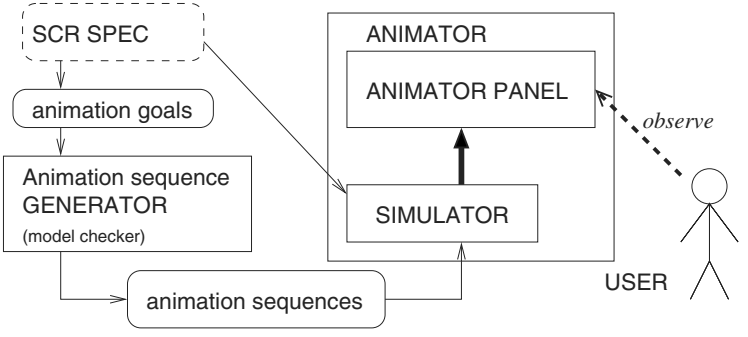


Fig. 1. Automatic model driven animation

3.1 Animation Goals

An *animation goal* is a formula that represents a particular behavior or property to animate. Formally, an animation goal is a predicate over a state or over a pair of states: the current one and the next one. To animate an animation goal a, we have to find a valid state sequence that ends with a state (or a pair of states) where a becomes true. We call this sequence of states *animation sequence* or animation scenario. For example, if the animation goal a is "$\mathtt{WaterPres} > 500$", we have to find a sequence of values for monitored variables such that the system reaches a state where `WaterPres` becomes greater than 500. Note that the notion of animation goal is very similar to test goal or test predicate as presented in [6,9], as well animation sequence is similar to test sequence.

For SCR specifications, we distinguish (as explained in Sect. 2) between requirements that refer to safety properties of the system, and functional requirements that are specified by tables and refer to the operation of the system. In the following, we explain how to derive animation goals in a systematic and automatic way from both safety properties and functional requirements. Note that this distinction is useful for the sake of clarity, but it would not be necessary, since functional requirements could be rewritten as properties.

Property Driven. Animation goals can be generated from safety requirements, and the corresponding animation sequences are useful to animate the system showing certain situations in which those requirements acquire particular importance.

We assume that safety requirements are formalized in Disjunctive Normal Form (DNF)[1]. Given a requirement $R = \bigvee_{i=1}^{n} C_i$, we define an animation goal a_i for each conjunct C_i, *i=1...n,* as follows.

Definition 1. *Given a property* $R = \bigvee_{i=1}^{n} C_i$*, we define animation goals for R, all the n formulas* $a_i \stackrel{def}{=} \bigwedge_{j \neq i} \neg C_j$*, i=1...n*

The animation goal a_i requires that all the conjuncts of *R*, except C_i, are false. Since *R* requires that at least a conjunct is true, the animation of a_i shows the behavior leading to a state where only C_i becomes true. For example, if the requirement *R* is $A \vee B$, we derive two animation goals: the first one is $\neg B$, leading to a state where only *A* is true, and the second one is $\neg A$, leading to a state where only *B* is true. If the requirement *R* has form $A \rightarrow B$, then it can be rewritten as $\neg A \vee B$, and the two animation goals for *R* are: *A* and $\neg B$. The animation sequence for the first animation goal leads the system to a state *s* where *A* becomes true and allows the user to check the validity of the implication, i.e. if *B* also holds in *s*. The animation sequence for $\neg B$ leads the system to a state *s* where *B* is false and allows to check if also *A* is false in *s*.

Example 1. Consider the property: $\texttt{WaterPres} < \texttt{Low} \rightarrow \texttt{Pressure} = \texttt{TooLow}$. The two animation goals for this property allow to animate a scenario leading to a state where `WaterPres` becomes less than `Low` in order to check that `Pressure` is equal to `TooLow`, and a scenario leading to a state where `Pressure` is not equal to `TooLow` to check if `WaterPres` is not less than `Low`.

Example 2. The two animation goals for the property (1) at page 299 are:

$@\texttt{T}(\texttt{WaterPres} < \texttt{Low})\ \texttt{WHEN}\ \texttt{Block} = \texttt{On} \ \wedge\ \texttt{Reset} = \texttt{Off},$
$\texttt{SafetyInjection}' \neq \texttt{Off}$

[1] A logical formula in DNF consists of a disjunction of conjunctions where no conjunction contains a disjunction. The generic format for a formula in DNF is $\bigvee_{i=1}^{n} C_i$. It is always possible to write a requirement in DNF. For example, the requirement $(A \vee B) \wedge C$ can be rewritten in DNF as $(A \wedge C) \vee (B \wedge C)$.

Remark 1. Since animation is used at the early stages of the system development, when heavier methods (like theorem proving) have not been applied yet to prove that the system satisfies a safety property *R*, one goal of animation is to find behaviors where *R* is not satisfied. Each animation goal a_j for *R* requires that all the conjuncts, except C_j, are false. Since *R* requires that at least a conjunct is true, *R* holds if C_j is true in the final state of the animation sequence for a_j. Otherwise, the animation sequence has uncovered a fault in the specification.

Functional Requirements Driven. Animation goals can be derived from the next state relation as specified by SCR tables. One animation goal is introduced for each cell in each table with the aim to separately animate the behavior specified by such cell. Formally, with reference to the notation given in Table 1, we introduce the following definitions.

Definition 2. *Given a condition table, for each condition* $c_{j\,k}$ *not equal to* `false`*, j = 1...n, k = 1...p, the animation goal for* $c_{j\,k}$ *is the formula* $a_{j\,k} \stackrel{def}{=} \texttt{Mode} = \texttt{m}_j \wedge \texttt{c}_{j\,k}$

Definition 3. *Given an event table, for each event* $e_{j\,k}$ *not equal to* `false`*, j = 1...n, k = 1...p, the animation goal for* $e_{j\,k}$ *is the formula* $a_{j\,k} \stackrel{def}{=} \texttt{Mode} = \texttt{m}_j \wedge \texttt{e}_{j\,k}$

Definition 4. *Given a mode table, for each event* $e_{j\,k}$ *not equal to* `false`*, j = 1...n, k = 1...*p_j*, the animation goal for* $e_{j\,k}$ *is the formula* $a_{j\,k} \stackrel{def}{=} \texttt{Mode} = \texttt{m}_j \wedge \texttt{e}_{j\,k}$

For each condition in condition tables, the animation goal has the aim of animating a scenario ending with a state where such condition is true, and, for each event in event and mode tables, a scenario ending with a state where such event occurs.

Example 3. The 5 animation goals for event table 3 defining `Overridden` are:

```
Pressure = High ∧ @F(Pressure=High),
Pressure = Normal ∧ @T(Block=On) ∧ Reset=Off,
Pressure = TooLow ∧ @T(Block=On) ∧ Reset=Off,
Pressure = Normal ∧ (@T(Pressure=High) ∨ @T(Reset=On)),
Pressure = TooLow ∧ (@T(Pressure=High) ∨ @T(Reset=On))
```

3.2 Generation of Animation Sequences

For automatic scenario generation, we use the method proposed in [6,9] which exploits the model checkers Spin [15] or SMV [20] and, in particular, their ability to generate counter examples. The method consists in the following steps. First, we encode the SCR specification in the language of the model checker (Spin or SMV) following the technique described in [1]; then, for each animation goal a_i, we compute the animation sequence that covers a_i by trying to prove with the model checker the *trap property* $\neg a_i$. If the model checker finds a state where $\neg a_i$ is false, it stops and prints as counter example a state sequence leading to that state. This sequence is the animation sequence for a_i. Note that the generation of animation sequences is totally automatic.

Infeasible Animation Sequences. The model checker always terminates and one of the following three situations occurs. The best case is when it stops finding that the trap property is false, and, therefore, the counter example to cover the animation goal is generated.

The second case happens when the model checker explores the whole state space without finding any state where the trap property is false, and, therefore, it proves $\neg a_i$. In this case, we say that the animation goal is *infeasible* or *not animable*. It is designer's responsibility to check whether this situation is due to a fault in the specification or not.

In the third case, the model checker 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 animation goal 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 designer that the animation goal 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 an animation sequence that covers a particular predicate is undecidable. Nevertheless, in our experience the third case is quite rare: for our case study it never happened.

Model Checking Limits and Benefits. Model checking applies only to finite models. Therefore, our method works for SCR specifications having variables with finite domains. However, this limitation does not preclude the application of our approach to models with infinite domains, thanks to abstraction techniques as described in [12]. 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.

Besides all these limits, the complete automaticity of the model checker allows to compute animation sequences without any human interaction.

3.3 Automatic Animation on Demand

Animation on demand is a particular variant of automatic model driven animation. This approach is pictured in Fig. 2.

In animation on demand, the user requires the animation of a particular behavior supplying the animation goal identifying such behavior, while the entire animation sequence is automatically computed starting from the model, as described in Sect. 3.2.

In Sect. 3 we have defined animation goals as predicates over the current and possibly the next state. They are derived from safety requirements or from conditions and events inside SCR tables. In animation on demand, the user can introduce new animation goals that can be complex temporal logic formulas expressing properties not only on the current

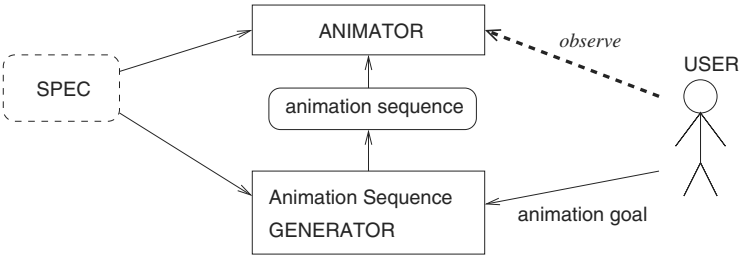


Fig. 2. Animation on demand

and next state but on a temporal sequence of events or conditions. We, therefore, extend the definition of animation goals allowing animation goals stated as CTL (Computational Tree Logic) or LTL (Linear Temporal Logic) formulas[2].

As an example, consider the case in which the user wants to simulate the following scenario: *system is not reset, then eventually the pressure of the water passes the* `Permit` *bound, then eventually it falls below* `Low`, *and then eventually the user blocks the safety injection*. Formally, the animation goal is the sequence of the following four conditions and events:

1. `Reset = Off`
2. `@T(WaterPres > Permit)`
3. `@T(WaterPres < Low)`
4. `@(Block = On)`

This sequence can be translated in CTL as:

(2)
```
Reset= Off & EF(WaterPres > Permit
                & EF(WaterPres < Low & EF(Block = On)))
```

being `EF` the CTL operator *eventually in the future*[3].

Once the user has written a particular animation goal, the model checker computes the animation sequence (if it exists) exploiting the use of the trap property (the negation of the animation goal) and the counter example generation. For example, SMV finds the counter example shown in Table 5 for the trap property of (2). The four conditions and events of the animation goal are written in bold.

Animation on demand is based on human interaction with the animator; however, differently from the user driven approach, users do not have to build the complete sequence of inputs, but just describe the scenario by an abstract logical formula. The model checker generates the animation sequence that satisfies the animation goal and the user animates the sequence to check if the behavior is correct. The drawback of this approach is that it requires some user skills in formulating a scenario as CTL or LTL formula. For this reason, the possibility of graphically building CTL or LTL formulas would be very useful (see Sect. 5).

[2] SMV is able to prove CTL formulas, while Spin uses LTL.

[3] The animation goal may contain other CTL temporal operators in order to state more complex formulas which are not necessary linear sequence of events and conditions.

Table 5. Animation sequence for (2)

state 1: *initial state*	state 3:	...
SafetyInjection = On	WaterPres = 5	state 96: *Pressure TooLow*
Permit = 1000	...	SafetyInjection = On
Low = 900	*WaterPres increases*	WaterPres = 998
Block = Off	...	Pressure = TooLow
Reset = On	state 95: *Pressure Permitted*	state 97: *SIS is blocked*
WaterPres = 2	SafetyInjection =Off	Block = On
Pressure = TooLow	WaterPres = 1001	Overridden = 1
Overridden = 0	Pressure = Permitted	SafetyInjection =Off
state 2: *SIS is reset*	...	
Reset = Off	*WaterPres decreases*	

4 An Animator Tool

In this section, we describe the general architecture of a prototype tool we have realized for automatic graphical animation of SIS. The tool architecture is depicted in Fig. 3.

We distinguish three main components of the animator:

1. the Tests Generator Tool: following user requests, it generates scenarios exploiting the model checkers and stores them in a repository; it is described in [6,9]
2. the Animator client: it retrieves generated scenarios and drives the animator service by providing the scenario to animate (i.e. the values of variables);
3. the Animator service: it shows the system status and the system behavior by means of a graphical animator panel.

Animator client and service have a three layers architecture. At communication level, client and server communicate through a CORBA ORB. At control level, the animator client controller and the animator server controller manage their graphical interfaces, start the processes, and connect and register themselves with the ORB. At GUI level, the animator client controls the animator through the control panel depicted in Fig. 4. The

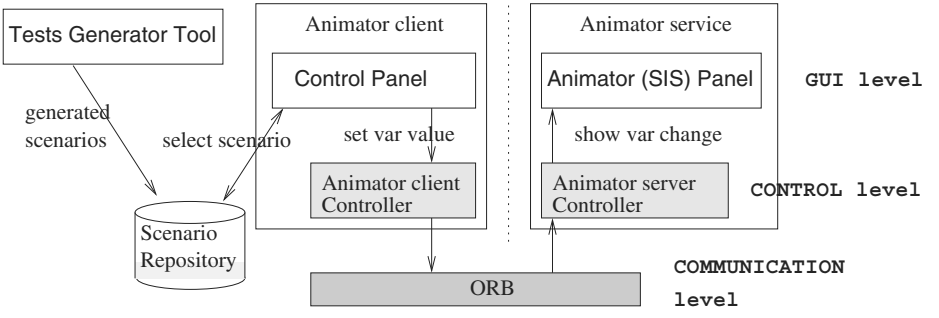
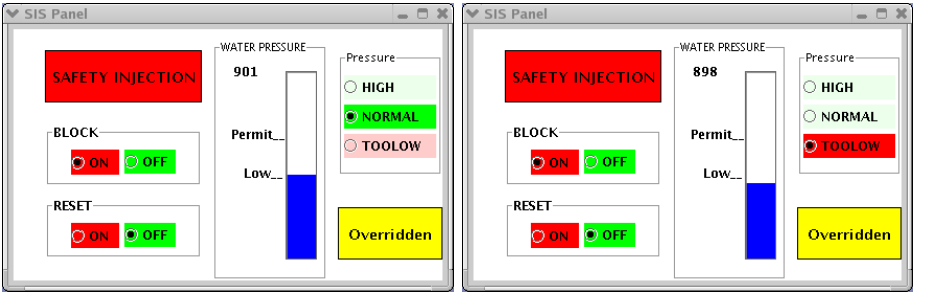
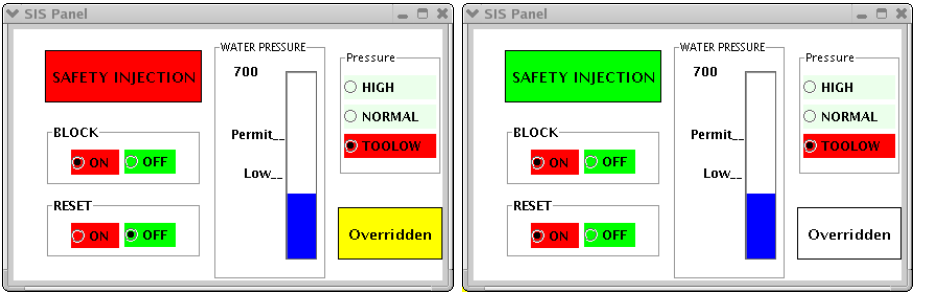


Fig. 3. Tool architecture

Fig. 4. Control panel



Animation of: `@T(WaterPres<Low)when Block =On and Reset =Off`



Animation of: `Pressure = TooLow and @T(Reset=On)`

Fig. 5. SIS animator panel

control panel provides two different ways to animate a scenario: step by step (the user makes the animator progress by pressing the next state button) or automatically (the user selects the time interval between two consecutive states). The user selects scenarios to animate from a scenario repository. A graphical interface that helps the user to link table cells and property requirements to scenarios is under development.

The GUI part of the animator service is the real graphical panel that shows the system state and behavior. We have built the SIS panel using the standard Java graphical library by means of the graphical form editor provided by NetBeans[4].

In Fig. 5, we show the last two states for two animation goals: one for the property 1 at page 299 and one for the event Table 3 at page 298. In the first case, `WaterPres` becomes less then `Low`, but the system is blocked and not reset, and, therefore, `Safety-`

[4] `http://www.netbeans.org`

`Injection` stays `Off`. In the second case, the system has `Pressure` `TooLow` and is reset (`Reset` becomes equal to `On`), and, therefore, `SafetyInjection` becomes `On`.

The architecture is highly modular and makes the animator easy to change. The animator panel can be easily substituted; this feature is relevant since each specification requires a new animator panel to be animated. Developing new animator panels is a matter of minutes using the form editor provided by NetBeans. Thanks to CORBA, the animator can work in a distributed way. The animator service may run on a remote machine, for example a computer of the customer, while the animator client would be controlled by the designer.

5 Related Work

There exist several tools and methods for animating formal specifications. [2] uses the B-Toolkit for animation of B specifications. The B-Toolkit presents the user a symbolic representation of the system state and allows the invocation of specification operations. The interface is mainly text based, and the user can perform queries and run commands by typing suitable instructions in a text console. This approach is more similar to simulation, because it does not exploit any graphical element. The approach presented in [16] suffers the same limitation. [16] clearly discusses the benefits of animation in the contest of light weight approaches to formal methods, in particular Z. The user can perform a set of queries checking the initialization, verifying the preconditions of schema, and performing a simple reachability property. The model checker Spin [15] has its own simulator that provides the user with information about the system state and allows, besides verification of properties, interactive simulation, simulation driven by counter examples, and random simulation. Also Spin displays this information mainly in text format. [22] proposes a random animation for Lustre specifications. Random test inputs are generated taking into account only the constraints about the environment. Safety requirements are checked using the generated scenarios. An AsmGofer[23] simulator for UML state machines execution is presented in [4]. The user has to execute the state machine and to query function values by a textual shell.

The use of a graphical domain-specific simulator for SCR is presented in [11]. SCR simulator supports the construction of graphical front-ends, tailored to particular applications. [11] presents a front-end for a real aircraft attack specification. A pilot, instead of entering values for monitored variables and seeing the values of the controlled ones, interacts with the simulator and the results are presented in the graphically simulated cockpit. A graphical simulator developed for the ASM specification of a light control system is presented in [3]. It is based on AsmGofer and uses TCL/TK for the animator panel.

A complex and complete graphical animator is presented in [19,18]. The authors develop an animator engine called Scenebeans based on Timed Automata semantics. They introduce *behavior beans* for actions and behaviors (for modeling system operations), as well as graphical components called SceneGraphs that represent the system state. A script language based on XML is introduced and used to build animations. Scenebeans is a flexible general purpose framework for animations. However, the problem of animation sequence generation in not tackled. In [18], Scenebeans is applied to an air traffic

control case study (Short Term Conflict Alert), and historical data are used as animation sequences.

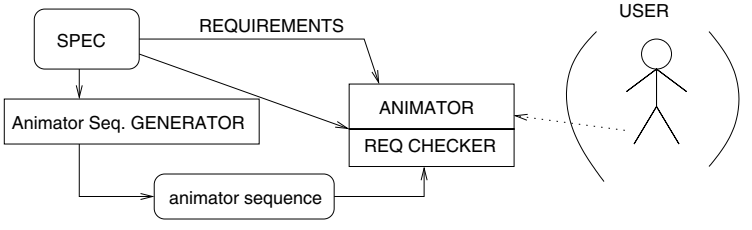
[21,10] present the use of the tool Possum to animate Z specifications. Graphical interfaces using TCL/TK can be easily implemented depending on the specification to animate. [21] presents a systematic approach to plan, document, and maintain animation scenarios starting from Z formal specifications. The user follows some guidelines to manually derive animation scenarios suitable to exercise the entire specification.

6 Conclusions and Future Work

In this paper we have presented automatic model driven graphical animation, a novel approach to animate requirements specification. Animation is useful to better understand requirements and to gain confidence of correctness of their specification. Automatic model driven animation minimizes user effort to build those scenarios able to animate all the critical system behaviors. We have introduced an approach to automatically generate animation sequences starting from SCR requirements specifications, and we have presented a prototype tool for the generation and animation of scenarios. In the future, we plan to work in several directions.

We plan to define some coverage criteria that can give a measure of how extensively the system has been animated. We plan to define new strategies to derive a greater set of animation goals starting both from SCR tables and from requirements. The user may want to split complex animation goals, to derive simpler animation sub goals, each animating a more particular critical behavior[5]. Another interesting issue is the animation of the *else* case[6] for event and mode tables [6].

A *requirements checker* can be integrated in the animator as shown by the following figure. It checks that all the safety requirements are never violated in animation sequences.



We plan to add the capability to graphically build complex animation goals. The user would compose CTL or LTL formulas by selecting events and conditions from a

[5] For example, the animation goal derived from event Table 3, `Pressure = Normal` $\wedge$ `(@T(Pressure=High)` $\vee$ `@T(Reset=On))` may be split in two different animation sub goals: `Pressure = Normal` $\wedge$ `@T(Pressure=High)` and `Pressure = Normal` $\wedge$ `@T(Reset=On)`.

[6] No events in the table occur and the variable does not change.

graphical panel. We also plan to support the translation of animation goals into natural language, in order the customer to better understand the meaning of animated behaviors.

Another important direction of future work is providing a graphical framework to build animator panels with ease. The user would choose graphical animator elements (buttons, lights, etc.) from a palette, connect them to specification variables, and place them on a pane. Initially, the palette would offer only a limited set of elements, but the user could introduce new graphical items. We plan to investigate the use of JavaBeans for this scope.

Automatic model driven animation can be transferred to other formal specification techniques for system behavior, e.g. to visual diagrammatic notations such as Petri nets, Statecharts or Message Sequence Charts, provided that encodings for such notations in the language of the model checkers exist. The definition of animation goals for such models will be subject of future work.

Acknowledgments. We thank the anonymous referees for their constructive comments.

References

1. R. Bharadwaj and C. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Software Engineering Journal*, 6(1), Jan. 1999.
2. J. Bicarregui, J. Dick, B. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29(1–2):53–78, July 1997.
3. E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, July 2000.
4. A. Cavarra and E. Riccobene. Simulating UML statecharts. In R. Moreno-Diaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science - Eurocast 2001*, pages 224–227, 2001.
5. P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993.
6. A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In O. Nierstrasz and M. Lemoine, editors, *Proceedings of the 7th European Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 1687 of *LNCS*, Sept. 6–10 1999.
7. A. Gargantini, L. Liberati, A. Morzenti, and C. Zacchetti. Specifying, validating and testing a traffic management system in the TRIO environment. In *Compass'96: Eleventh Annual Conference on Computer Assurance*, page 65, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.
8. A. Gargantini and A. Morzenti. Automated deductive requirements analysis of critical systems. *ACM Transactions on Software Engineering and Methodology*, 10(3):255–307, July 2001.
9. A. Gargantini and E. Riccobene. ASM-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11):1050–1067, Nov. 2001.
10. D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *Thirteenth International Conference on Automated Software Engineering*, pages 302–305. IEEE Computer Society Press, 1998.

11. C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR: A toolset for specifying and analyzing software requirements. In *Proc. 10th International Computer Aided Verification Conference*, pages 526–531, 1998.
12. C. Heitmeyer, J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, Nov. 1998.
13. C. L. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering, Two Volumes*. John Wiley & Sons, January 2002.
14. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, April–June 1996.
15. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
16. E. Kazmierczak, M. Winikoff, and P. Dart. Verifying model oriented specifications through animation. In *Asia Pacific Software Engineering Conference*, pages 254–261. IEEE Computer Society Press, 1998.
17. R. A. Kemmerer. Testing formal specifications to detect design errors. *IEEE Transactions on Software Engineering*, 11(1):32–43, Jan. 1985.
18. J. Magee, J. Kramer, B. Nuseibeh, D. Bush, and J. Sonander. Hybrid model visualization in requirements and design: A preliminary investigation. In *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10)*, Nov. 2000.
19. J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, June 2000.
20. K. L. McMillan. The SMV system. Technical report, Carnegie-Mellon University, Pittsburgh, PA, 1992. DRAFT.
21. T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. In *Proc. of the 2001 Australian Software Engineering Conference (ASWEC 2001)*, pages 76–85. IEEE Computer Society, 2001.
22. I. Parissis. A formal approach to testing lustre specifications. In *1st International IEEE Conference on Formal Engineering Methods, Hiroshima*, pages 91–100, 1997.
23. J. Schimd. Executing ASM specifications with AsmGofer. `http://www.tydo.de/AsmGofer`.

Probe Mechanism for Object-Oriented Software Testing

Anita Goel[1], S.C. Gupta[2], and S.K. Wasan[3]

[1] University of Delhi, Dyal Singh College
New Delhi-110003, India
aagoel@vsnl.com
[2] National Informatics Center, NIC, A Block
New Delhi-110003, India
scgupta@hub.nic.in
[3] Jamia Millia Islamia, Department of Mathematics
New Delhi-110025, India
skwasan@yahoo.com

Abstract. This paper presents a probe-based testing technique that facilitates observing internal details of execution at different levels of abstraction-unit, integration and system levels, during testing of object-oriented software. Our technique adapts probe, an observability measure, to suit the testing needs of object-oriented software. It uses source-code instrumentation, which requires probes to be pre-determined and pre-built in the software during the development phase. Test coverage reports are generated from the information gathered by the executed probes. It includes coverage of probes at probe, method, class, inheritance, regression and dynamic binding levels. During regression testing, our technique helps in the selection of test cases that must be re-executed. Furthermore, the log generated by active probes can be used for post-analysis.

1 Introduction

The unique architecture and features of object-oriented software introduce new kind of errors in the software. As a result, some issues involved in the testing of object-oriented software are different from the conventional software testing issues [1, 2]. In order to handle the unique testing issues of object-oriented software, the conventional software testing techniques require improvisation or new ones need to be developed.

During testing, a correct output result does not always ensure correctness of processing. An incorrect state may not be reflected in the output, but it governs the future behavior of methods. An incorrect output will have to be diagnosed in terms of the various execution steps. This requires access to the internal behavior [20].

Object-oriented software is tested at unit, integration and system levels. Class is the basic unit of testing. Class is composed of data structure and a set of methods. Objects are run time instances of the class. The data structure defines the state of the object that is modified by the methods defined in the class. The encapsulation feature of object-oriented software hides the data structure, posing difficulty in accessing the

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 310–324, 2003.

state of the object, which is essential for verifying the correctness of processing and for error diagnosis. The already tested units integrated via relationships like inheritance and aggregation are tested during integration testing. The focus is on the interaction among the units. System testing at input/output level requires information about the interaction among the integrated units.

We see that the focus of testing shifts as we move from unit to system testing. Correspondingly, the focus of internal execution details observed during testing must also shift as we move from unit to system testing. Several techniques exist [5, 12, 14, 15, 16] that verify/display the state of the object. Tools exist that provide method-level or statement-level trace [7, 11] of execution details or allow specific values to be observed [6, 9, 13, 18] during testing. But, none of them address the issue of observing internal execution details at different levels of abstraction during testing,

In this paper, we focus on observing internal execution details at different levels of abstraction- unit, integration and system levels, during testing of object-oriented software. During unit testing, the input/output of the methods and the impact of method execution on the state of the object are observed. The sequence of execution of classes and input/output of the class is observed during integration testing. The input/output of the integrated units is observed during system testing.

Here, we present a probe-based testing technique that adapts *probe-an observability measure* for object-oriented software testing. It uses source-code instrumentation. Probes are pre-determined and pre-built in the software during design and coding phases, for observability needed at different levels of abstraction. During testing, probes are externally activated/deactivated, facilitating visual display of execution details at unit, integration and system levels. Probes can be turned off when internal execution detail is not needed. Probes left embedded in software results in creation of *testable software* for further modifications and easy *corrective maintenance*.

The coverage of inheritance hierarchy and dynamic binding relationship is needed to ensure adequate testing of these relationships. Our technique uses the internal execution details to generate test coverage of probes at probe, method, class, *inheritance and dynamic binding* levels. Due to the iterative and incremental nature of object-oriented software, regression techniques are needed during development phase as well as during maintenance. Our technique helps to identify the test cases to be re-executed during *regression testing.* It also generates coverage of the changed unit.

A probe-based testing tool based on the probe-based testing technique has been developed. It is implemented in Java and designed using use case driven object-oriented approach. The probe-based testing technique has been applied to test UIServer-a translator software that translates UIML (User Interface Markup Language) document to WML (Wireless Markup Language) or CHTML (Compact- HyperText Markup Language) document. UIServer has been developed using Java and XML.

In this paper, Section 2 discusses probe- an observability measure. Section 3 describes the probe-based testing technique. In section 4, the probe-based testing technique during testing phase is discussed. Section 5 describes in brief, the experience in testing of UIServer software using our technique. Section 6 describes related work. Section 7 states the conclusion.

2 Probe - An Observability Measure

Observability measures are provisions in the software that facilitate observation of internal and external behavior of the software, to the required degree of detail [20]. The need to build observability in object-oriented software has been stressed by Binder [17]. According to Binder, "if you cannot observe the output of a component under test, you cannot be sure how a given input has been processed".

Traditionally, print statements and debuggers have been used, to get access to internal information. Print statements require frequent commenting, uncommenting and recompilation of code each time changes are made. As we move from unit testing to system testing, deciding what to comment/uncomment becomes difficult. It requires intimate knowledge of the software. Moreover, print statements are not structured, posing hindrance in the analysis of internal information details. Debuggers give access to all the information, at a point in execution. But what is important to observe becomes harder and harder to decide, as the size of the software increases.

The concept of probe is an effective observability measure for observing internal details of the software. Probe is a method invocation having a structured probe identifier and a message [20]. The syntax of probe method invocation is *probe(probe_id, probe_message)* where, *probe* is a method name, *probe_id* is a unique structured identifier identifying the location of *probe_message* and *probe_message* contains state-related attributes, parameter values, temporary variables or a message, relevant at the location of probe in the code. Probes are inserted in the software at locations where information relevant at that point is needed. During execution of software, probe can be activated/deactivated and probe breakpoints can be set, externally. An active probe on execution generates a message carrying the internal information along with its identity. Probes can be turned on/off as and when required. Probes do not interfere with the logic of the software and can be left embedded in the software.

3 Probe-Based Testing Technique

The probe-based testing technique facilitates observation of internal execution details during testing of object-oriented software. The execution details consist of the-

Class being executed
Method of the class being executed
Value of parameters or messages at method entry/exit or within the method
Hierarchy of execution of classes
Probe identification to locate probe displaying parameter/message, in the code

These execution details are displayed at unit, integration and system levels. The idea behind our testing technique is quite simple. Our technique uses probe, an observability measure, to observe the internal details of execution during testing. Probes are inserted in the source-code during software development. During testing, probes are controlled externally– activate/deactivate, to display execution details at unit, integration and system levels. A point to be noted here is that our technique gathers internal details of execution from the probes inserted in the software. Thus, the content

of the probe must be carefully decided and probes must be inserted at proper locations. Our technique is defined in three phases -

(1) *Probe Structure* defines the structure of *probe_id* and *probe_message*.
(2) *Probe Insertion* defines locations in the software where probes are to be inserted.
(3) *Probe Subsystem* defines the different subsystems that operate on the software embedded with probes, during testing.

Using our technique requires- defining the probes, inserting probes in the software and using probe subsystem during testing. The software developer should have knowledge of our technique during the design and coding phases.

We illustrate our technique with an example of link list, written in Java, shown in Program code 1(relevant details shown). For the time being, we do not focus on the "Log" statements. The program is one integrated unit consisting of three classes (1) UserInterface (2) LinkList (3) Node. UserInterface accepts input from the user to add/delete to link list. LinkList has methods to create list, add/delete nodes from the list. Node has methods to create node with string data, get and set data etc. LinkList creates a link list, where each node is of type class Node. We extend this example to create heterogeneous link list, to show coverage at inheritance and dynamic binding levels (code not shown). Class IntNode is derived from the class Node. IntNode creates node with integer data. A node in link list is of type IntNode or Node. IntNode inherits getNext() and setNext() from Node and defines its own methods getData(), setData(), printData() etc. The method printData() is dynamically bound.

Program code 1. Example of Link List (relevant details shown)

```
class UserInterface{
public static void main(String[] args) {
1.      Log d = new Log("UserInterface");
2.      Log.penter("L1/1", "msg:start main");
        LinkList lst = new LinkList();   // Create a list
             :
             :
    // get string from user to be inserted in the list
    // lst.addNodeAtHead(new String(str));
    // get string to be deleted from the list
    // lst.deleteNode(new Node(str));
             :
        lst.printList();
3.      Log.pexit("L1/5", "msg:end main");
4.      Log.close();  } }
class LinkList{
        private Node head;
public LinkList(){...}                          // head = null
public Node getHead(){...}                      //returns head
public void addNodeAtHead(String s){...}
                              //adds node with data s at head
public void printList(){...}              //prints link list
public void deleteNode(Node n){
        Log.penter("L2/9", "delete node=" + n);
        if (head == null) {
            Log.pexit("L2/10","msg:list empty");
```

```
            return;}
        if (n.equals(head)) {
            head = head.getNext();
5.          Log.pmsg("L3/11","msg:node deleted at head");}
        else { Node p = head, q = null;
               while((q = p.getNext()) != null) {
                 if (n.equals(q)) {
                    p.setNext(q.getNext());
                    Log.pmsg("L3/12","deleted node=" + q);}
                 else p = q;  }}
        Log.pexit("L2/13","msg:end delete node");
        return; }
class Node  {
        private String data;
        private Node next;
public Node(String wrd){...}     //initialises data = wrd
protected Node getNext(){...}        // returns next node
protected void setNext(Node n){...}      //sets next to n
public String getData(){...}        //returns string data
public void setData(String s){...}       //sets data to s
public void printData(){...}                //prints data
public boolean equals(Object o){
        Log.penter("L2/13", "compare=" + o);
        boolean bool = false;
        if (this == o)  {
             Log.pmsg("L3/14", "msg:object equal");
             bool = true; }
        else if (!(o instanceof Node)) {
            Log.pmsg("L3/15", "msg:object not equal");
            bool = false; }
        else if ((((Node)o).data).equals (this.data)) {
            Log.pmsg("L3/16", "msg:data equal");
            bool = true; }
        else { bool = false;
               Log.pmsg("L3/17", "msg:not equal"); }
        Log.pexit("L2/18", "comparison was= " + bool);
        return bool;  } }
```

3.1 Probe Structure

Our technique associates level number with the probe, to display execution details at different levels of abstraction. Also, each probe must be uniquely identifiable, to locate probe in the code. Probe structure defines *probe_id* as- "*level_number/probe_number"* where, *level_number* indicates the testing level. The *level_number* is L1, L2 and L3 for probes that display interaction among the integrated units (system level testing), interaction among the classes of integrated unit (integration level testing) and code level behavior of a class (unit level testing) respectively. The *probe_number* uniquely identifies a probe in a class. For each class, it starts from 1 for the first probe and is incremented for every other probe in the class. It may be in any order within the class. E.g. "L2/2" represents an integration level probe having probe number 2.

The structure of *probe_message* is defined as *"variable1:val variable2:val ..variableN:val* msg*:string"* where, val is the value of a variable, and, msg displays a message. E.g. "msg:start main" is interpreted as, *start main* is a message.

The designer and code developer decide the *level_number* and *probe_message* respectively, during the design and coding phases of software development.

3.2 Probe Insertion

Our technique requires probes to be inserted at the beginning, end (before return statement) and anywhere between begin and end of the method (if needed) like in if-statement or loop statements, as shown in Program code 1, line 2, 3, 5 respectively. Probe insertion is the responsibility of the developer.

The methods are of three kinds in object-oriented software- public, protected and private. Public methods are invoked from outside the class. Private methods are invoked by public methods. Protected methods are private to the class but can be invoked from the derived classes. The *level_number* in *probe_id* is decided based on the kind of method and the location of probe in the method. Probes defined at the beginning and end of a method have *level_number* L1 in the public methods of classes that interact with other integrated units, L2 in the public methods of rest of the classes, and, L3 in private and protected methods of a class. Probes defined anywhere in between the beginning and end of any method has *level_number* L3.

To display the hierarchy of execution of classes and the parameter values at method entry/exit, probe insertion defines static methods-- *penter, pexit and pmsg,* for probes inserted in the beginning, end and anywhere in between a method respectively. Probes are inserted as *Log.penter, Log.pexit, Log.pmsg*, as shown in Program code 1, line 2, 3, 5 respectively. The class "*Log*" defined in our technique interacts with the software embedded with probes during testing. To start and stop recording information from probe, line 1 and 4 in Program code 1 are needed respectively.

3.3 Probe Subsystem

The probe subsystem work on the software embedded with probes, during testing. It has four components –Preprocessor, OnLineTest, OffLineTest, and Report. ***Preprocessor*** works on the compiled software before its execution. Its functions are- (1) store details of inheritance hierarchy- classes, declared and inherited methods of each class, and class to which inherited method belongs. (2) Store details of dynamic binding relationship- classes, the dynamically bound methods and method from where the dynamically bound methods are invoked. (3) Store "*level_number/ probe_number/class_name/tag/method_name*" for each probe. It is needed to display the execution details during testing. The *level_number* and *probe_number* are defined in probe_id. *Tag* is probe method *penter, pexit* and *pmsg*. The *class_name* and *method_name* is the class and method respectively, in which the probe is defined. To get class executed during testing, Preprocessor inserts "*class_name*" in probe_id of probes. (4) Insert "*getClass()*" in *probe_message* of probes of inherited methods, to find the class invoking the inherited method. The function *getClass()* must be sup-

ported by the programming language. It is supported in Java. (5) Identify modified classes and help in selection of test cases to be re-executed, during regression testing. ***OnLineTest*** is invoked on execution of the software, for testing. It defines probe settings for probe activation and probe breakpoint to facilitate observation of internal execution details at unit, integration and system levels. Output of active probes is stored in *log file*. Output of all executed probes is also stored in a file. ***OffLineTest*** is invoked after testing, to analyze the log file and to get details about the software-classes, methods etc. ***Report*** is invoked after testing to generate coverage report.

4 Using Probe Subsystem

The components of probe subsystem facilitate observation of internal execution details at unit, integration and system levels, selection of test cases to be re-executed during regression testing, post-analysis of log file and report generation.

4.1 Probe Settings

Probe settings allow the tester to externally control the probes during online and off-line testing. **Probe Activation** defines commands to selectively activate/deactivate probes, externally, during testing. Output of only active probes is displayed on the screen and stored in the log file. Probes are referred to in a generic style so that a group of probes can be addressed through a single command. The format of probe activation command to activate and deactivate probe is

A: *class_name/method_name/level_number/probe_number* (1)

D: *class_name/method_name/level_number/probe_number* (2)

respectively. The *level_number* in (1) and (2) results in activation of probes at the specified *level_number* and at lower *level_number*, and, deactivates probes at the specified *level_number* respectively. E.g. "A:*/*/L2/*" activates all probes having *level_number* L1 and L2 in all classes of all methods. The command "D:Node/*/L3/*" deactivates all probes having *level_number* L3 in all methods of class Node. A '*' used for *level_number, class_name* or *method_name* matches with all *level_number, class_name* and *method_name* respectively.

Probe BreakPoint allows breakpoints to be set on selected probes. On occurrence of break, execution of the software pauses. The tester can observe the already displayed probes and change the probe settings to observe rest of the execution details. Probe breakpoint can be set as follows:

class_name/method_name/level_number/probe_number (3)

A String (4)

In (3), break occurs when the *level_number, class_name, method_name* and *probe_number* are true in the probe being executed. In (4) break occurs when output of a probe contains the specified string. E.g. string "start deleting" results in a break when it is encountered in the probe output.

Probe settings are made during testing based on what is to be observed and at what level of detail. All probe settings are stored in a file and can be modified during testing. All commands in the file are applied sequentially, to find active probes.

4.2 OnLine Testing

During testing, activating/deactivating the *level_number* in probe activation command facilitates observation of internal execution details at different levels of abstraction. The *class_name, method_name* and *probe_number* can be activated or deactivated to "fine-tune" the execution details to be observed during testing. Probe breakpoint is set to observe the already displayed probes.

Table 1. Internal execution details at unit level (delete node from list) of Program code 1 (->enter, <-exit, --between)

Class Name	Method Name	LevelNo./Probe No.
ListClient	main(String[])	L3/3 msg: Start deleting
|→Node |←Node	Node(String)	L2/1->msg: create string node L2/2<-data=stringB
|→List	deleteNode(Node)	L2/9->delete node=Node@f133f325
|→Node	equals(Object)	L2/13->compare=Node@2debf324
|--Node	equals(Object)	L3/16 msg: data equal
|←Node		L2/18<-comparison was=true
|→Node |←Node	getNext()	L3/3->msg: get next node L3/4-> next=Node@e96ff324
|--List	deleteNode(Node)	L3/11 msg: node deleted at head
|←List		L2/13<-msg: end delete node

Unit Testing: Probes at *level_number* L3 are activated during unit testing. It results in activation of probes at *level_number* L1, L2 and L3. The internal execution detail of deleting a node from the link list, at unit level, is shown in Table 1. It displays code level behavior of the program- the executed public, private and protected (*getNext()*) methods, the part of if-condition executed (*msg:data equal*) and a message in between the method (*msg:node deleted at head*), at begin and end.

Integration Testing: During integration testing, probes at *level_number* L2 are activated. It results in activation of probes at *level_number* L1and L2. The internal execution detail of link list at integration level is shown in Table 2. It displays sequence of execution of classes and their input/output. E.g. for node deletion it displays that the node was compared, found to be equal and deleted. It does not display the internal execution details of the class as shown during unit testing in Table 1.

Table 2. Internal execution details at integration level of Program code 1

Class Name	Method Name	LevelNo./Probe No.
ListClient	main(String[])	L1/1->msg: start main
|→List |←List	List()	L2/1->msg: List() L2/2<-head=null

|→List	addNodeAtHead(String)	L2/5->add node=stringA
|→Node	Node(String)	L2/1->msg: create string node
|←Node		L2/2<-data=stringA
|←List	addNodeAtHead(String)	L2/6<-added at head=Node@e96ff324
|→List	addNodeAtHead(String)	L2/5->add node=stringB
|→Node	Node(String)	L2/1->msg: create string node
|←Node		L2/2<-data=stringB
|←List	addNodeAtHead(String)	L2/6<-added at head=Node@2debf324
|→Node	Node(String)	L2/1 ->msg: create string node
|←Node		L2/2 <-data=stringB
|→List	deleteNode(Node)	L2/9->delete node=Node@f133f325
|→Node	equals(Object)	L2/13->compare=Node@2debf324
|←Node		L2/18<-comparison was=true
|←List	deleteNode(Node)	L2/13<-msg: end delete node
|→List	printList()	L2/14->msg: print list
|→Node	printData()	L2/11->msg: print string data
|←Node		L2/12<-data=stringA
|←List	printList()	L2/15<-msg: list printed
ListClient	main(String[])	L1/5<-end main

System Testing: Probes at *level_number* L1 are activated to observe interaction among the integrated units of the system.

Corrective Maintenance: It involves fixing reported errors in released and in-use software. For this, there is a need to understand the software to identify the component containing the error and to locate erroneous code in it. Since maintenance team is generally not same as development team, understanding the software is difficult due to its large size and complexity. The erroneous code cannot be isolated using the limited information available at user interface.

Our technique helps in the corrective maintenance activity. The software is re-executed with probes on, with input that resulted in error. Observing the internal execution details at system level, integration level and unit level helps to identify the integrated unit, the class of the integrated unit, and, the method and probe of the class displaying incorrect value, respectively. The *probe_number* in the probe displaying incorrect value is used to locate the probe in the class. Code near the probe is examined to isolate erroneous piece of code.

4.3 Regression Test Cases

Regression testing is the process of validating modified software to detect whether new errors have been introduced into previously tested code and to provide confidence that modifications are correct [8]. An overview of regression testing techniques is given in [8][21]. Here, we focus on selection of regression test cases. Having known the unit (method, class etc.) of change (add, delete, modify), there is a need to select the test cases to be re-executed, from the already executed test cases.

Our technique identifies class as the unit of change. It includes classes in the hierarchy, if changed class is part of inheritance hierarchy. Preprocessor identifies the test cases to be re-executed in two steps. First, it identifies the test cases containing the

changed class name, from the file storing all executed probes. It includes classes impacted by changed class, since the impacted class invokes it or is invoked by it. Next, from this subset, test cases containing unique sequence of *probe_number, class_name* (invoking the changed class - of the changed class - invoked by changed class) is identified. The resulting test cases are to be re-executed.

4.4 OffLine Testing

OffLineTest facilitates analysis of log file using probe setting defined in OnLineTest. Probe settings allow the tester to "play" with the contents of log file. The behavior of a class or a method can be observed/analyzed. For very large software, execution details of a portion of the software can be observed.

Table 3. Information about the Program code 1

Class Name	Method Name	LevelNo./Probe No
ListClient	main(String[] args)	L1/1 L3/2
	:	
	:	
Total Classes=3	Total methods=13	Total Probes=36

OffLineTest also provides details of the software- (1) classes, methods and probes. (2) Inheritance hierarchy, inheritance category, and, declared and inherited methods of each class, and (3) Dynamic binding relationship. It also provides count of these relationships, as shown in Table 3. It retrieves these details from Preprocessor.

4.5 Report Generation

All probes traversed (active/inactive) are logged, which, help to generate coverage of probes at- probe, method, class, inheritance, regression and dynamic binding levels. The logged probes can also help to identify covered path sequences, for test data adequacy. Coverage reports can be viewed as *%coverage* and *in the form of list*, for- *a single, set* and *all test cases combined*. The *list of uncovered probes* generated at these levels helps to locate untested code. To calculate coverage, Report gathers details of the executed and total probes from OnLineTest and Preprocessor respectively.

Probe, Method and Class Coverage: The coverage of probes at probe, method and class levels is displayed as (*class_name,probe_number1,…,*), (*class_name, method_name1,…*) and (*class_name*) respectively, as shown in Table 4.

Table 4. Coverage at probe, method and class levels of Program code 1

Uncovered Probes	Uncovered Methods	Uncovered Classes
(Node, 7, 8, 9, 10, 14, 15, 17) (List, 10, 12,)	(Node, getData(), setData(String))	None
%coverage: 75.0%	% coverage: 84.61539%	%coverage:100.0%

Inheritance Coverage: The creation or modification of an inheritance hierarchy falls in four categories [1][2]-

(A) Creation/Modification of superclass
(B) Creation/Modification of subclass
(C) Modification of superclass data structure from subclass, and,
(D) Pure extension of superclass.

Testing for category

(A) and (C) necessitates retesting of superclass, all subclasses and units dependent on it

(D) requires testing of subclass only, and,

(B) requires testing of subclass and retesting of inherited methods of superclass. Retesting is needed because inheritance provides new context for the inherited methods. The inherited methods, thus, must be executed from the class in which it is declared and from the class inheriting it.

Our technique calculates coverage at inheritance level based on inheritance category. Inheritance coverage is displayed as (*probe_number/class_name/method_name /class invoking the inherited method*). The *class invoking the inherited method* gets value from *getClass()* (inserted by Preprocessor). It is blank if method is executed from the class in which it is declared. As shown in Table 5, getNext(), setNext() defined in class Node are invoked from Node itself, statement (1), (3) and from IntNode (inheriting class), statement (2), (4).

Table 5. Coverage at inheritance level (of inherited methods)

3/Node/getNext()	(1)
3/Node/getNext()/IntNode	(2)
5/Node/setNext()	(3)
5/Node/setNext()/IntNode	(4)

Dynamic Binding Coverage: Testing a dynamic binding relationship requires testing of all possible methods that can get bind at runtime for a single method invocation. Coverage at dynamic level is displayed as (*probe_number/class_name/ method_name/method_name from where invoked*). The "*method_name from where invoked*" is the method enclosing call to the dynamically bound method. As shown in Table 6, printData() defined in Node and IntNode, is invoked from method printList().

Table 6. Coverage at dynamic binding level (as list of covered probes)

11/Node/printData()/printList()
8/IntNode/printData()/printList()

Regression Coverage: Coverage at regression level is displayed as coverage at probe level of the modified unit. It includes coverage of probes at inheritance level if modified class is part of the inheritance hierarchy.

5 A Case Study - UIServer

The probe-based testing technique has been used during testing of an XML based software-UIServer, written in Java. UIServer operates in the web environment. It translates a UIML (User Interface Markup Language) document to WML (Wireless Markup Language) or CHTML (Compact-HyperText Markup Language) document.

UIServer consists of three integrated units- Validate, ParseTree and GenerateTML. *Validate* receives URL of the UIML document and the http request-UAData (UserAgentData) from the web engine. It fetches the UIML document, checks its validity according to UIML 2.0a DTD (Document Type Definition- it defines the structure of UIML document. It specifies the tags, attributes of these tags and their arrangement with each other. UIML document can contain 28 tags.). Validate returns valid/invalid UIML document. If document is valid, *ParseTree* is invoked. It parses the UIML document and builds a tree based on the tag hierarchy defined in UIML2.0a DTD. It stores the tags defined in UIML document as nodes of the tree. If no error found, GenerateTML is invoked. *GenerateTML* uses the tree to generate WML/CHTML document as the output. In case of error, an error message is output.

Our technique was used to observe the internal execution details at different levels of abstraction during the testing of UIServer. We discuss an instance, where without our technique, locating the erroneous code was a difficult task. During system testing, for a UIML document as input, the WML document generated was incorrect. No error message was displayed. Observing the internal execution details at system level displayed error in the output of the integrated unit- ParseTree. Next, the execution details observed at integration level, displayed error in the output of class ParseUIML, of ParseTree. While observing the internal execution details of ParseUIML at unit level, it was found that the code developer had not taken any action for the tag <attribute> defined in the tag <call> of UIML document. The error was notified.

Table 7. Characteristics and test execution result of UIServer

#of lines of code	#of probe inserted	Execution time with probes on	Execution time with probes off	Coverage at probe level	Coverage at method level	Coverage at class level
8994	705	5.85s	5.60s	84%	88%	100%

Table 7 shows characteristics of UIServer and test execution result using our technique. It includes number of lines of code and probes in the software, execution time with probes on and off, and coverage of probes at probe, method and class level. The uncovered probes helped to locate the untested portion of code.

6 Related Work

This work is related to observability measures in object-oriented software and testing of object-oriented software.

6.1 Observability Measures

Previous research focuses on observing state of the object during testing of object-oriented software. Assertions, [5][12] are injected at key locations in the source code and state-space monitored. Assertion monitoring does not display internal execution details as long as assertions evaluate to true. It only verifies state of the object. False assertion evaluation triggers an exception. McGregor and Korson [14] emphasize functional testing and provide observer methods to check externally observable states. Murphy et al. [16] provide state-reporting methods with every class. Dorman [15] uses friend declaration to observe private data of C++ program. To check method call sequence, it inserts callback functions before and after the call.

Probes are used in tools to trace execution details or observe values of specific variables during testing. Compaq's JTREK, JOIE [6] and BIT [9] use byte-code instrumentation for troubleshooting Java applications. It requires watch points to be inserted in the software, to observe values of selected parameters, return values etc. The selective instrumentation, based on what needs to be observed is a limitation for testing. It requires understanding of the internal behavior of the software. JIE [11] and instr [7] use source-code instrumentation of Java programs for method-level tracing, test coverage, execution logs, debugging etc. But, the trace needed to understand the behavior of software, itself needs to be understood owing to its large size.

Probes are also used for profiling in compiler optimization feedback and performance tools. Hundt [18] describes HP Caliper, which uses dynamic instrumentation to provide a framework for building tools for performance analysis, coverage analysis and correctness checking. DeRose et al. [13] describe- Dynamic Probe Class Library (DPCL) an infrastructure for developing instrumentation for performance tools.

Tools like Panorama JavaTest, CodeWork JCover and TestWorks' TCAT instrument the code to generate test coverage report at class, method, statement and branch levels. They also display the executed and the unexecuted part of the code.

6.2 Testing Object-Oriented Software

Unit Testing: Most of the research work in class testing is based on generation of method sequences as test cases, execution and result validation. Souter et al. [3] develop a code-based testing and analysis tool– TATOO to generate test tuples based on object manipulation. Chen et al. [10] develop a tool TACCLE that selects fundamental pairs of equivalent ground terms and checks the observational equivalence of resulting objects. Turner et al. [5] use automata to model internal representations of the class, generate test cases using state-based test suites and verify the final state.

Integration testing: According to Chen et al. [10], little study has been made on cluster-level testing or its relationship with class-level testing. They perform static cluster testing on horizontal interactions among classes using individual message-passing rule. Jin and Offutt [22] generate test cases using coupling based criteria.

Dynamic binding: Alexander and Offutt [19] extend coupling-based testing to detect faults from polymorphic relationships among components. Orso et al. [4] extend traditional data flow test selection criterion. They define def-use sets to select

execution paths that can reveal failures due to incorrect combination of polymorphic calls. Payne et al. [12] implement inheritance hierarchy such that pre and post conditions made in base class are not violated by polymorphic methods.

Inheritance Testing: Dorman [15] uses call-interface instrumentation to determine coverage of inherited methods of base class in context of their usage. Payne et al. [12] use down-call technique to restrict inheritance to be a sub-typing relationship. Retesting of base class in context of the derived class is not needed.

7 Conclusion

In this paper, we present a probe-based testing technique that adapts probe-an observability measure for object-oriented software testing. It facilitates observation of internal execution details at different levels of abstraction- unit, integration and system levels, during testing. Our technique requires probes to be pre-determined and prebuilt in the software during design and code phases of software development. During testing, probes are externally activated/deactivated facilitating observation of internal details of execution at unit, integration and system levels. Coverage of inheritance hierarchy and dynamic binding relationship generated using our technique helps in determining the adequacy of testing these relationships. Our technique also aids in the selection of regression test cases. Probes can be turned off when the internal execution details are not needed. Probes left permanently embedded in the software results in creation of testable software for further modifications and easy corrective maintenance.

Acknowledgement

We are grateful to Dr. Mukul Sinha, Director, Expert Software Consultants Limited, for his valuable comments and suggestions in the development of the testing technique and application of our technique to UIServer project. We wish to thank Tanmoy, Pawan and Ramakant for their help in the implementation of the concepts discussed in this paper.

References

1. A. Goel, S.C. Gupta, K. D. Sharma: Object Oriented Testing: An Overview. In: Proceedings of International Conference on Software Engineering And Its Application, 91-99, Hyderabad, India, Jan (1997)
2. A. Goel, S.C. Gupta, S. K. Wasan: Object Oriented Software Testing: A Survey Report. In: Proceedings of 2nd Annual International Software Testing Conference, QAI, Bangalore, India, Jan (2001)
3. A. L. Souter, T. Wong, S. Shindo, and Lori L. Pollock: TATOO: Testing and Analysis Tool Object-Oriented Software. In: Proceedings of 7th International Conference, Tools

and Algorithms for the Construction and Analysis of Systems, held as part of ETAPS, April (2001)
4. A. Orso, M. Pezze: Integration Testing of Procedural Object-Oriented Programs with Polymorphism. In: Proceedings of 16th International Conference on Testing Computer Software, Washington D.C., June (1999)
5. C. D. Turner, D. J. Robson: A State-Based Approach To The Testing Of Class-Based Programs. Software Concepts and Tools, Vol. 16, No. 3, 106-112, (1995)
6. G. A. Cohen, J. S. Chase, D. L. Kaminsky: Automatic Program Transformation with JOIE. In: USENIX Annual Technical Symposium, 167-178, (1998)
7. Glen Mc Clunskey & Associates LLC: Java source code instrumentation. http://www.glenmccl.com/instr
8. Graves, Harrold, Kim, Porter, Rothermel: An Empirical Study of Regression Test Selection Techniques. In: ACM Transactions on Software Engineering and Methodology, Vol 10, No. 2, 184-208, April (2001)
9. H. B. Lee, B. G. Zorn: BIT: A Tool For Instrumenting Java Bytecodes. In: Proceedings of USENIX Symposium on Internet Technologies and Systems, 73-82, Monterey, California, Dec (1997)
10. H. Y. Chen, T. H. Tse, T. Y. Chen: TACCLE: A Methodology for Object-Oriented Software Testing at the Class and Cluster Levels. ACM Transactions Of Software Engineering And Methodology, Vol. 10, No. 4, 56-109, Jan (2001)
11. Java Instrumentation Engine: http://dl.tromer.org/jie
12. J. E. Payne, R. T. Alexander, C. H. Hutchinson: Design-for-Testability for Object Oriented Software. Object Magazine, SIGS Publications Inc., Vol. 7, No.5, 34-43, (1997)
13. Luiz DeRose, Ted Hoover Jr., J. K. Hollingsworth: The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. In: Proceedings of International Parallel and Distributed Processing Symposium (IPDPS), April (2001)
14. McGregor, Timothy D. Korson: Integrated Object oriented Testing and Development Processes. Communication of ACM, 59-77, Sept (1994)
15. Misha Dorman: C++ "It's Testing Jim, But Not As we Know It. In: Proceedings of EuroSTAR, Edinburg, Scotland, Nov (1997)
16. Murphy, Paul Townsend, Pok Sze Wong: Experiences with Cluster and Class Testing. Communication of ACM, 39-47, Sept (1994)
17. Robert V. Binder: Design for Testability in Object Oriented Systems. Communication of ACM, 87-101, Sept (1994)
18. Robert Hundt: HP Caliper- An Architecture for Performance Analysis Tools. In: Proceedings of WIESS , San Diego, California, USA, Oct (2000)
19. R. T. Alexander, A. J. Offutt: Analysis Techniques for Testing Polymorphic Relationships. In: Proceedings of TOOLS, Santa Barbara, California, USA, August (1999)
20. S. C. Gupta, M. K. Sinha: Improving Software Testability by Observability and Controllability Measures. 13th World Computer Congress, IFIP, 94, Vol 1, 147-154, (1994)
21. Yuejian Li, N J Wahl: An Overview of Regression Testing. Software Engineering Notes, ACM SIGSOFT, 69-73, Jan (1999)
22. Z. Jin and A. J. Offutt: Coupling-based Criteria for Integration Testing. Journal of Software Testing, Verification and Reliability, Vol. 8, No. 3, 133-154, Sept. (1998)

Model Checking Software via Abstraction of Loop Transitions

Natasha Sharygina[1] and James C. Browne[2]

[1] Carnegie Mellon University, 5000 Forbes Ave., Pittsburgh, PA, USA 15213
nys@sei.cmu.edu
[2] The University of Texas, Austin, TX, USA 78712
browne@cs.utexas.edu

Abstract. This paper reports a data abstraction algorithm that is targeted to minimize the contribution of the loop executions to the program state space. The loop abstraction is defined as the syntactic program transformation that results in the *sound* representation of the concrete program. The abstraction algorithm is defined and implemented in the context of the integrated software design, testing and model checking. The loop abstraction technique was applied to verification of NASA robot control software. The abstraction enabled model checking for realistic robot configurations where all other state space reduction approaches, including BDD-based verification, predicate abstraction and partial order reduction, failed.

1 Introduction

Formal verification by model checking has the potential to produce a major enhancement in software reliability and robustness for software systems. The applicability of model-checking to software systems is severely constrained by "state space explosion". Data abstraction is a principal method for state space reduction [1,4,6,13,14,16]. Predicate abstraction [7] is one of the most popular and widely applied methods for systematic abstraction of programs. Predicate abstraction is based upon abstract interpretation [5]. It maps concrete data types to abstract data types through predicates over the concrete data. However, complete predicate abstraction may be intractable due to its computational cost. Generation of a full set of predicates is typically infeasible for large programs.

All forms of abstraction may introduce unrealistic behaviors (behaviors not found in the concrete program) into the abstract program. Error traces from model checking of the abstract program are often used to rule out unrealistic behaviors. Excessive abstraction may introduce additional behaviors which result in state space explosion when attempting model checking for the abstract program. These drawbacks for general abstraction methods coupled with the potential effectiveness of abstraction, motivate research into targeted abstractions which can be applied selectively.

This paper formulates and evaluates an abstraction algorithm that minimizes the contribution of the loop executions to program state space. The loop

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 325340, 2003.

abstraction generates an abstract program with the same static task graph as the concrete program from which it is derived but which specifies a minimum (or nearly minimum) number of traversals of the loops of the static task graph. These abstract programs have orders of magnitude smaller state spaces that the concrete programs from which they are derived. We demonstrate that the algorithm is *correct* in that the "abstract" program is a conservative approximation of the "concrete" program with respect to the control specifications of the program. The correctness result implies that a control specification holds for the original program if it holds for the abstract program. Some loss of precision of data computations introduced by the abstraction is traded for the ability to conduct *practical* verification of behavioral specifications of control algorithms. The potential usefulness of loop abstraction is enhanced by the facts that control software are the obvious candidate systems for model checking to improve reliability and that almost all control systems implement feedback loops.

The properties of the loop abstraction algorithm are:

- It is computationally simple and requires storage linear in the size of the program since it is a source to source transformation based on static analysis of the program.
- It is based on syntactic manipulation of expressions, and produces a reduced program and therefore, it can be applied without change to the verification tool or the verification algorithm.
- It produces a syntactic representation of the abstract program and thus other model-checking state space reduction techniques, such as symbolic model-checking and partial order reduction, can be applied to the abstract program.

The loop abstraction algorithm has been implemented in the integrated high-level design (xUML) and automata-based model checking software development framework (Fig. 1) and has been evaluated during verification of a NASA robot controller. It has been found to give order of magnitude reduction in the complexity and computational resource requirements for model-checking of control properties of a robot control system. Most importantly, the loop abstraction enabled completion of model checking for realistic robot configurations where all other approaches, including predicate abstraction [1,15], failed.

Contents of Paper. Section 2 defines a framework for the project. Section 3 defines the program syntax and semantics. Section 4 presents the loop abstraction algorithm. The effectiveness of loop abstraction is demonstrated in Sect. 5 that shows the verification results of the NASA robot controller system. Section 6 summarizes the paper and gives an overview of related work.

2 Integrated Design and Verification Framework

The loop abstraction algorithm has been defined in the context of the software development framework that integrates xUML modeling[1], testing and automata-

[1] xUML is a dialect of UML with executable semantics. Programs written in xUML are *design level* representations which can be executed directly through discrete event

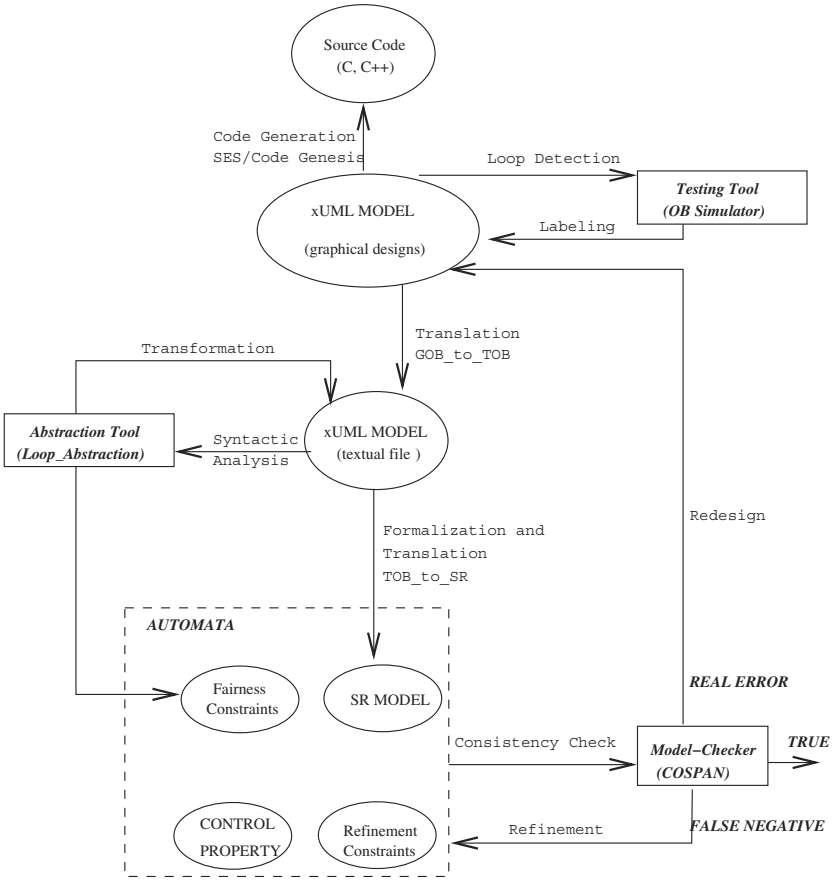


Fig. 1. The integrated design and model-checking software development environment

based model-checking as shown in Fig. 1. We refer the reader to [19,22] for a detailed description of the integrated environment.

The framework consists of the following components:

1. *The xUML graphical specification and validation environment as it is implemented in the commercial tool,* SES/ObjectBench (OB) [17].

An xUML program is a set of interacting objects. The behavior of each object is implemented as a *Moore state machine* with a bounded FIFO input queue for events. The objects interact by sending and receiving events. Each state of the state machine which can receive an event is given a unique label. A sequential action is associated with each labeled state. Each action assigns values to state variables and generates events to be posted to its own input queue or the input

simulation or interpretation and/or compiled to procedural source code. xUML is fairly widely used for development of control systems [11,17]. A complete specification of the xUML notation can be found in [20,21].

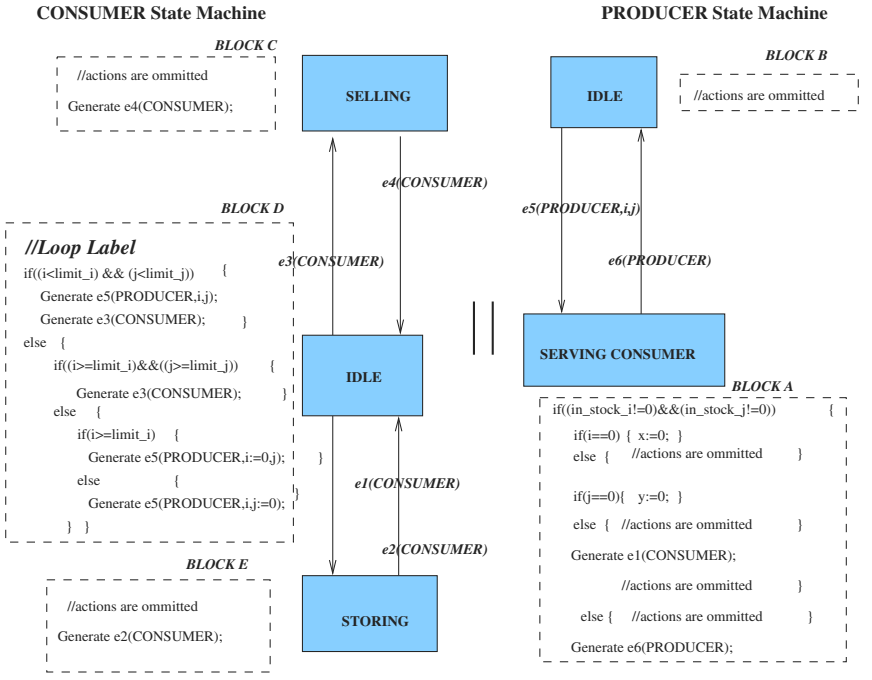


Fig. 2. The *consumer-producer* xUML program

queues of other state machines. The actions execute in run to completion mode. The action language for the implementation of xUML is a C-based language extended by the event generation and state machines manipulation commands. The execution model for an xUML system is asynchronous interleaved execution of the action language programs associated with the labeled states of the state machines. An example of the xUML system is given using a *Consumer-Producer* xUML program (Fig. 2). The sample program is modeled by xUML state machines, representing behavioral specifications of the *Consumer* and *Producer* xUML objects.

Each state machine is represented as a collection of *blocks* that are activated by events. For example, *Block D* of the *Consumer* state machine represents a block of actions that can be activated by an input event *e4* or *e2* and labeled by the update of a variable *status := IDLE* (the label variables update statements are implicitly implemented by the xUML graphical development environment.) The activation of the block is followed by the execution of local statements for variables update and generation of output events *Generate e3(CONSUMER), Generate e5(PRODUCER,i,j), Generate e5(PRODUCER,i:=0,j)* and *Generate e5 (PRODUCER,i,j:=0)*. Note, the distinction between fields of the event *e5*: different data is passed by the event depending on the satisfaction of the specified conditions. For example, if at some point during the program execution a variable

i is larger or equal to some predefined value, *limit_i*, than the event *e5* will pass a zero value to the *Producer* process, using the first supplemental data field of the event command.

2. *The* LOOP_ABSTRACTION *program.* This component of the integrated environment is the subject of the paper. The LOOP_ABSTRACTION program implements the loop abstraction algorithm which is outlined in Sect. 4. The *loop_abstraction* program takes as an input results of the program behavioral analysis conducted using the discrete event simulator. The event simulator is a part of the xUML specification and validation environment. During the simulation the program is executed by traversing possible *event sequences* which can arise from the execution of interacting xUML state machines. The set of actions that are repeatedly initiated by some event are manually annotated with a *Loop Label* in the xUML specification environment. An example of the annotation is shown in Fig. 2, *block D*.

The *loop_abstraction* algorithm results in the syntactic program transformation of the original program that maps all traversals of a loop in the program control flow defined by different values of the program variables to traversals with values of *path_selection* variables (see Sect. 4.2) which abstract values uniquely specify each distinct event sequence within the loop.

3. *The automata-based model-checking tool,* COSPAN[2]. Consistency check is performed over the abstract SR model (SR is the input language of COSPAN) automatically derived from the abstract xUML program with respect to the specified control property, the set of fairness constraints and the approximation restrictions[3]. The following features provided by COSPAN are used to support the loop abstraction procedure:

- the *assume/guarantee mechanism* of COSPAN is used to add fairness constraints and refinement assumptions to the model-checking process.
- the *localization reduction* algorithm, automatically invoked by COSPAN during model-checking, is used to eliminate from consideration variables that do not effect the verification property.

3 Background

3.1 Program Syntax

Control software systems are often constructed as *compositions of sequential programs* which interact through sending of events, $S = p_1 \parallel \ldots \parallel p_n$. Each program, p = *(X,E,I,B)* is defined as a set of variables, X, a set of events, E, an initial condition, I, and a set of *basic blocks* (defined next), B, that contain commands

[2] A detailed description of COSPAN and its features can be found in [8,13].

[3] The abstraction procedure uses a translator [22] that automatically transforms the xUML programs from the Graphical OB representation into SR, an input language of the model-checker, COSPAN. Specifically, the LOOP_ABSTRACTION program is applied to the intermediate representation of the translation result, the textual representation of the xUML programs.

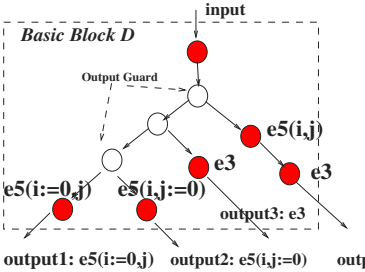
that modify the program variables, and send and receive events. For example, an *assignment* command, $x := any\{\ exp_1,...,exp_n\ \}$ is a *non-deterministic* assignment, after which a program variable, x, will contain the value of one of the expressions $exp_1,..., exp_n$; *'Generate e(ID,exp)'* is a *communication* command that sends an event, e with some data, exp, to the destination program identified by its name, ID[4].

Definition 1 [Basic Block] (cf. [9]). *A* basic block *is a sequence of statements for which execution can be initiated only through the statement at the head of the block and which, once initiated, executes to completion.* Execution of a basic block is initiated by arrival of an event.

Events are distributed via FIFO queues, one queue for each sequential program. The execution model for a *sequential program* is: a) An event arrives in the input queue of a sequential program and some basic block of the program is enabled for execution in "run to completion" mode. b) The enabled basic block is executed. c) Execution of a basic block may result in events being sent to the program containing the executing basic block or to other programs. d) At the end of the execution of a basic block the program halts and awaits arrival of its next event.

Definition 2 [Output of a Basic Block]. *The* output of a basic block *is an event or sequence of events.* The output from an instance of the execution of a basic block is determined by the control structure within the block. Each instance of the execution of a basic block is a traversal of the tree determined by the control structure. The control statements which generate the tree will be referred to as the *block output guard*. The outputs of a basic block are determined by the leaves which are reached in the execution of the block. Thus, each branch of the block output guard controls *one* output of a block.

Figure 3 illustrates the concept of the basic block. The control flow graph (at




```
int i, j, limit_i, limit_j;
Basic Block D:
    Receive e2(CONSUMER);
    if((i<limit_i)&&(j<limit_j)) {
            Generate e5(PRODUCER,i,j);
            Generate e3(CONSUMER);      }
    else {  if((i>=limit_i)&&(j>=limit_j)) {
            Generate e3(CONSUMER);      }
        else {  if(i>=limit_i) {
                Generate e5(PRODUCER,i:=0,j);      }
              else {
                Generate e5(PRODUCER,i,j:=0);      }
            } }
```

Fig. 3. Demonstration of a basic block concept

[4] For the complete list of the commands see [17]

the command level) illustrates the control flow paths that determine the outputs of the basic block.

The execution model for the system is *asynchronous interleaved* execution of the basic blocks of the sequential programs. a) One program from among those which are enabled for execution (those programs with events in their input queues) is non-deterministically selected for execution. b) The basic block in the selected program which consumes the event at the head of the event queue is executed and step *a* is repeated.

Definition 3 [Basic Block Control Flow Graph]. *The nodes of* the basic block control flow graph of the system are *basic blocks* of the composing sequential programs. *The arcs* of the basic block control flow graph of the system connect basic blocks which are the sources and targets for *events*. Therefore a control flow graph can also be specified as generation and consumption of a sequence of events.

The control flow properties of the system behavior can be stated in terms of control at the basic block level by referring to events that initiate execution of basic blocks.

This works exploits the *atomicity* of the program executions to identify loops of the program execution (loops across basic blocks).

Definition 4 [Loop]. *A* loop *in a basic block control flow graph of a system is defined by a repeated execution of a path which begins with the generation of a unique event by a basic block and ends at that same basic block (loop basic block, B^{loop}).*

Each loop is guarded by a set of the basic block output guards of the loop basic block, B^{loop}, and their dependence set, B^{depend}. The dependence set is the set of basic blocks which output guards operate on variables that are dependent on the output guard variables of the loop basic blocks. Let us call the variables of the basic block output guards that define the loop, *loop control variables*.

3.2 Program Semantics

The syntax of the program defined above can be given an execution semantics as an asynchronous transition system (ATS) [10] composed of finite state machine interacting through finite, non-blocking FIFO queues.

Definition 5 [Event Queue]. (cf. [10]) *An* event queue, $Q_i = (V,N,E,L)$ is defined by the *the queue vocabulary*, V, by the *size of the queue*, N, by the *vector of events stored in the queue*, E, and the *content of the stored events*, L, defined as a finite set of the values. The values are expressions on the system variables, or constants. For a set of queues, $\mathcal{Q}$, the queues vocabularies are disjoint.

Definition 6 [Finite State Machine]. (cf. [10]) *A* state machine, *M, is defined as a tuple,* $M = (X,S,s_0,I,O,\mathcal{Q},T)$, where

- *X is the finite set of variables*;

- *S is the finite set of possible binding of values to X*;
- s_0 *is* an element of S, *the initial state*;
- *I is the set of input events*;
- *O is the set of output events*;
- $\mathcal{Q}$ *is a set of event queues*;
- *T is the transition relation specifying the allowed transitions among S.*

Definition 7 [Trace of a State Machine]. *An infinite sequence of states* $tr = s_0 s_1 ... s_n$, is a trace of FSM *if (1)* s_0 *is an initial state and (2) for all* $0 \leq i < n$, *the state* s_{i+1} *is a successor of* s_i.

Definition 8 [Asynchronous Transition System (ATS)]. (cf. [10]) *An* ATS *is a composition of finite state machines which interact by sending and receiving events.* The global state space is the product of the local state spaces of the composed state machines, the system event queue is the union of the sets of the queues of the separate machines, and the global transition relation is the union of the local transition relations.

Definition 9 [Trace of an ATS]. *The trace of an ATS is an interleaving of states from the traces of the state machines which compose the system.* The ATS may be constrained by *fairness constraints* that determines which traces of the model are confronted with the specification during model checking. If a fairness constraint is defined as set of states, then a *fair trace* must contain an element of each fairness constraint infinitely often.

Definition 10 [Refinement]. Let A and C be two ATS instances. Let L(A) and L(C) be the language of all traces from execution of A and C.

If $X^C \subseteq X^A$, and L(C) $\subseteq$ L(A) then C *weakly refines* A, C $\leq$ A.

Definition 11 [Control Refinement]. Let us define an operator R which projects from L(C) and L(A) all states which do not receive events. Call R.L(C) and R.L(A) *control traces* of an ATS.

If $X^C \subseteq X^A$ and R.L(C) $\subseteq$ R.L(A) then C *weakly refines control* of A.

Control refinement is defined to show behavioral correspondence between the control (event) sequences of the abstract and concrete programs. The program actions are grouped into basic blocks (as defined earlier) and execute in run to completion mode. Therefore R.L(C) and R.L(A) correspond to the basic block control flow graphs of systems C and A and L(C) and L(A) correspond to the traces of systems C and A.

Definition 12 [Control Property]. *A* control property *is a linear temporal logic specification defined over states that input events.* For example, in Fig. 2 an event *e1(CONSUMER)* accepted by the *CONSUMER* program defines a control state of the program ATS.

4 Loop Abstraction

The execution behaviors of control software systems are typically dominated by cycles implementing feedback loops. The structure of the control flow graph is usually determined by a small set of variables (control flow variables). The paths in the control flow graph of a program with loops are usually determined by conditional statements (guards) which depend on a subset of the control flow variables (loop control variables). Model checking of such systems generates a traversal of the loops in the control flow graph for each possible value of each loop variable. Each traversal of the loop with different values of the loop control variables is distinct in the state graph of the program. Additionally each traversal of a loop will typically involve many variables ("don't care" variables) which do not participate in determination of the paths through the control flow graph. But each execution of a loop with different values for the "don't care" variables is also distinct in the state graph generated by the model checker.

Control flow properties (such as guarantee that the program components will never execute an unsafe sequence of control actions) are dependent only on the control flow graph of the system and are independent of the number of traversals of the loops of the control flow graph. Therefore the control properties of the concrete program can be model checked by model checking of an abstract program with the same control flow graph.

The abstraction presented in this paper generates an abstract program with the same static task graph as the concrete program from which it is derived but which specifies a minimum (or nearly minimum) number of traversals of the loops of the static task graph. The values of the "don't care" variables can also be freed in the abstract program. These abstract programs typically have orders of magnitude smaller state spaces than the concrete programs from which they are derived.

4.1 Sketch of the Loop Abstraction Algorithm

The algorithm iteratively analyzes and transforms basic blocks that define control flow within a loop[5]. The algorithm computes a number of outputs for each block that is executed in the control loop and uses the computed data to abstract the events generation within the basic blocks from the actual data. The steps in the loop abstraction algorithm are defined below. Each step is accompanied by a narrative description of the performed actions. The detailed description of the algorithm can be found in [18].

a) Identify each control flow statement (simple or compound) which participates in determining a path of a loop.

This step is implemented by identifying the output guards of the basic blocks that are repeatedly activated. The loop abstraction algorithm starts from a loop

[5] The basic block structure of the xUML programs is explicitly preserved by special words *'state'* and *'endstate'* in the textual representation of xUML programs for the beginning and the end of the basic block respectively. This allows syntactic identification of the code that corresponds to each xUML basic block.

basic block annotated by *'Loop Label'* during simulation of the program executions (see Sect. 2 for details).

b) Determine the number of exit paths from each control flow statement that determines the loop.

The algorithm conducts a trivial syntactic analysis of the basic block code and determines the number of branches of the block output guards that control output events. For example, there exist *four* outputs controlled by an output guard of basic *block D* of the *Consumer-Producer* example (see Fig. 3).

c) Replace each block output guard with a control flow statement with the same set of exit paths where exit path selection is determined by a variable (*path_selection* variable) whose range is defined by the number of outputs (computed in step b) controlled by each block output guard. The values of the *path_selection* variables are non-deterministically chosen. The transformation algorithm is trivial and is performed by copying statements and replacing the conditions of the block output guard by equality comparison of the *path_selection* variable to one value in its range. There are several patterns of the possible configurations of the control tree defined by the block output guard. The transformation algorithm resolves each pattern accordingly such that each transformed output guard truly represents the original program structure. An example of an output guard transformation for the basic *block D* from the *Consumer-Producer* example is given below. The right side represents the original text of the basic block and the left side demonstrates the result of the syntactic transformation.

```
Abstract Basic Block                    |   Concrete Basic Block

path_selection := any(1,2,3);           |if((i<limit_i)&&(j<limit_j)) {
if(path_selection == 1) {               |  Generate e5(PRODUCER,i,j);
  Generate e5(PRODUCER,i,j);            |  Generate e3(CONSUMER); }
  Generate e3(CONSUMER);  }             |else {
else {                                  | if((i>=limit_i)&&(j>=limit_j)){
if(path_selection == 2){                |   Generate e3(CONSUMER); }
    Generate e3(CONSUMER);  }           | else {
else {                                  | if(i>=limit_i){
  if(path_selection == 3){              |   Generate e5(PRODUCER,i:=0,j);}
     Generate e5(PRODUCER,i:=0,j);}     | else {
      else {                            |   Generate e5(PRODUCER,i,j:=0);}
     Generate e5(PRODUCER,i,j:=0);}     |     }  }
     } }                                |
```

d) Identify all of the variables which depend on the variables which appeared in the control statements that determine the loop. For example, local variables i and j of the *PRODUCER* program used in the *block A* are the dependent variables of the abstracted variables of the *CONSUMER* program's *block D*.

e) Identify all of the control flow statements which are defined over the variables detected in step d. For example, *if* control statements defined over the local variables i, j of the basic *Block A* are such control statements.

f) Replace these control flow statements following steps b) and c).

The algorithm terminates when all control flow statements defined over the loop control flow variables and their dependency set are detected.

In addition to the program analysis and transformation, the loop abstraction algorithm automatically creates a file that is used to store the *fairness assumptions* specified as one of the results of the program transformation. Fairness constraints are specified for each value of each *path_selection* variable. For example, during transformation of the *Block D* of the *Consumer-Producer* program, the following set of the fairness constraints is created[6] *Assume Eventually path_selection*:= 1; *Assume Eventually path_selection*:= 2; *Assume Eventually path_selection* := 3. The file containing the fairness constraints is passed to the model checker and used to specify assumptions (using the *assume-guarantee* features of the model checker, COSPAN) that assure that all outputs defined in the concrete system are explored during the model-checking of the abstract program.

The abstraction is enforced by the syntactic transformation of the basic blocks that are executed in the control loop. The formal definition of the abstract program is $P^a = (X^a, E^a, I^a, B^a)$, where E^a, I^a are defined as for the concrete program, $X^a = X \cup X^{new}$, where X is the set of variables defined in the concrete program and X^{new} is the set of the *path_selection* variables; $B^a = B \setminus B^{loop \triangleleft depend} \cup B^{new}$, where B, $B^{loop \triangleleft depend}$ are the sets of concrete basic blocks such that B^{loop} is the loop basic block and B^{depend} is its dependency set (as defined in Sect. 3.1); and B^{new} is the set of the transformed B^{loop} and B^{depend} basic blocks.

4.2 Soundness of the Loop Abstraction

We demonstrate that the loop abstraction is sound with respect to the control flow representation of the concrete program. The soundness result implies that a control specification holds for the original program if it holds for the abstract program.

Let C be an ATS instance associated with the concrete program. Let A be an ATS instance associated with a program that was constructed from C by applying the loop abstraction algorithm.

Theorem 1. *Given a control property □, the abstract ATS (A) is equivalent with respect to □ to the original ATS (C).*

Proof Sketch: The claim is proved by a trace containment test. We demonstrate that a control trace which conforms to the specification of the control property (see def. 12) of C is contained in the language of control traces, *R.L(A)*.

1. $X^C \subseteq X^A$. This follows the definition of the abstract program (see Sect. 4.1): (during program transformation new variables, the *path_selection* variables, are added to the program).

2. $E^C = E^A$. This follows the definition of the abstract program (see Sect. 4.1): (during program transformation *no* new events are added to list of the program events nor are any events of the concrete program are omitted).

[6] The assumptions are encoded in a query language of COSPAN.

3. Generation of events is controlled by the output guards. Call the variables that are used in the output guards of the concrete program, control variables, $X_{control}$. Call the rest of the variables of the concrete program, data variables, X_{data}. Therefore, $X = X_{control} \cup X_{data}$.

The *path_selection* variables are the control variables of the abstract program since they are used in the guard statements to control generation of events. Therefore, from 1 it follows that $X^C_{control} \subseteq X^A_{control}$.

4. Assume that the language of control traces is defined by a set of control traces each of which is initiated by valuation of a different control variable. Let's call these control traces *elementary* control traces.

From 2 and 3 it follows that any *elementary* control trace of C is a subset of *R.L(A)*.

5. From definition of traces (def. 7), every prefix of a trace is a trace. Since the set of initial states is not empty, and the transition relation is serial, every trace can be extended. Therefore, a control property (def. 12) is a specification that is defined over a set of elementary traces. Thus, from 4, any control trace conforming to a control property specified for C is contained in *R.L(A)*.

Therefore, A is equivalent to C with respect to the control property.

It can be shown in the manner above that R.L(C) $\subseteq$ R.L(A) which implies (see def. 11) that C weakly refines control of A and preserves *all* control properties. This means that the same abstraction can be used to check all control properties.

5 Evaluation of the Loop Abstraction Technique

The loop abstraction technique has been evaluated during verification of a NASA Robot Controller System (RCS) formulated as xUML programs. Due to the space limitations we present partial results of the RCS verification and refer the reader to [12,18,19] for the detailed description of the RCS and its properties.

Sample properties (both safety and liveness) are given in Table 1. The properties are encoded in a query language of COSPAN and refer to variables of the RCS xUML programs. For example, declaration *p* refers to the *abort_var* variable of the *Controller* RCS xUML program, *fk* refers to the *forward_kinematics* variable of the *EndEffector* RCS xUML program, and *r* refers to the *ee_reference* variable of the *Trajectory* RCS xUML program.

Table 1. Verification properties

N	Property	Robotic Description	Formal Description
1	EventuallyAlways(*p*=1)	Eventually the robot control terminates	Eventually permanently *p*=1
2	NeverUntil(*r=1*,*fk=1*)	No command to move the robot arm is scheduled before an initial position of the arm is computed	It is never the case that *r=1* holds until *fk=1* holds

Table 2. Comparison of verification of the concrete and abstract robotic systems

i	P1: Concrete (states/m:s/MB)	P1: Abstract (states/m:s/MB)	P2: Concrete (states/m:s/MB)	P2: Abstract (states/m:s/MB)
2	2.2e+12/350:4/735	26K/0:28/4.03	2.3e+11/344:4/713	17K/0:17/3.38
3	3e+18/415:4/1,246	63K/3:10/4.9	2e+17/410:3/1,190	63K/3:10/4.9
4	6e+23/592:4/1,802	145K/11:28/8.4	6e+24/662:3/2,190	116K/7:03/7.1
5	M/T exhaustion	688K/28:10/23.9	M/T exhaustion	554K/13:40/19.1
6	M/T exhaustion	1.1M/42:17/96.5	M/T exhaustion	715K/33:17/36.2

We considered several variants of the RCS of different complexity defined by the number of joints i of a robot arm (i defines degrees of freedom (DOF) of the robot arm). We used two types of programs to check the properties. The first type is the complete (concrete) program. The second type is the abstract version of the concrete program to which the loop abstraction method has been applied.

Table 2 compares the run-time and memory usage for properties from Table 1. The results are given for the concrete and the abstract RCS with a total number of 7 xUML programs. They exclude the i programs corresponding to the number of instances of the *Joint* object. Each entry in the table has the form $x/y/z$ where x is the number of the states reached, y is the run-time in cpu minutes and seconds (m:s) and z is the memory usage in Mbytes (MB). The results of the verification demonstrate significant reduction in both time and space for the abstract program compared to the concrete program. The results are given for the explicit state space exploration experiments and demonstrate that the reduction becomes more pronounced for larger values of i. Verification for the robot configurations consisting more than 4 joints (4 DOF) could not be completed for the concrete program due to the memory/time exhaustion (denoted as *M/T exhaustion* in Table 2), but COSPAN *succeeded for the abstracted model*. It is notable that application of the state space reduction techniques such as BDD-based verification and partial order reduction to verification of the concrete program also resulted in the state space explosion for programs implementing control of robots with more than 4 DOF. Our efforts on the predicate abstraction of the concrete program also failed. Specifically, the predicate abstraction technique supported by COSPAN [15] did not succeed due to the memory exhaustion during computation of the abstraction predicates. The boolean abstraction approach [1] resulted in over-approximation of the program executions that led to the state space explosion during verification of the abstracted program even for programs with less than 5 DOF. Thus, the loop abstraction technique became the *only* approach that enabled verification of robot configurations higher than 4 DOF.

6 Conclusions, Future and Related Work

Conclusions. The paper presented an approach for *practical* model checking of large-scale software. A loop abstraction technique has been defined and implemented in the context of the integrated design and model checking software

development. The abstraction algorithm is computationally simple and requires storage linear in the size of the program since it is a source to source transformation. It proved to be highly effective in state space reduction for the test-case control-intensive program, the large-scale robot controller system. Most importantly, the loop abstraction enabled completion of model checking for realistic robot configurations where all other approaches, including predicate abstraction [1,15], failed.

It would be expected that a selective and limited scope abstraction such as the loop abstraction would introduce fewer unrealistic behaviors into the abstract program than more comprehensive abstractions. This proved to be the case for the robot control system. Only a few refinements were needed. These were identified as false negatives in model checking the abstract program and were manually implemented. Propagation of the abstraction across basic blocks, planned as a future work, will further reduce the number of unrealistic behaviors introduced by the abstraction and hence the requirement for refinement.

A limitation of the loop abstraction is that it can only be applied when the properties to be model checked are control properties. Control properties are, however, typically the safety-critical properties of control systems.

Related Work. Loop abstraction is similar to predicate abstraction in that it requires specification of an abstraction function as predicates over concrete data. Loop abstraction differs from predicate abstraction in that it does not require computation of the abstraction predicates. Instead it operates on the conditional predicates which implement program control. The result of the loop abstraction is the construction of a control skeleton which makes our work similar to construction of boolean programs as defined in [1]. However, our work is different from [1] in that it is concerned with the abstraction of only the loops. Loop abstraction introduces a *limited number* of unrealistic behaviors compared to [1] and also preserves some original data valuations compared to the complete data abstraction provided by predicate abstraction methods. Loop abstraction can be a useful complement to predicate abstraction. It abstracts control while predicate abstraction abstracts statements not effected by the loop abstraction.

The implementation of the loop abstraction algorithm is similar to [15] in that the loop abstraction algorithm does not construct the explicit state graph of either the original or of the abstract program. Instead a syntactic analysis of the original program is used to produce an abstract program. However, our approach is different from other abstraction algorithms dealing with the source code in that the abstraction is applied to a design-level specification (xUML programs). To our knowledge, there has been no previous reports on data abstraction algorithms specifically targeting design level specifications.

The work presented in this paper is also related to path coverage (also known as predicate coverage) testing [2,3]. Path coverage reports whether each of the possible paths in each function of the program has been followed. (A path in testing is a unique sequence of branches from a function entry to exit). Loop abstraction provides complete coverage of all possible execution paths within a loop. One of the major obstacles to successful path coverage is looping during program

execution. Since loops may contain an unbounded number of paths, path coverage only considers a limited number of looping possibilities. Our method solves this problem. Path coverage has the problem that many potential paths are impossible to reach because of data relationship constraints. Loop abstraction algorithm solves this problem by adding fairness constraints to force exploration of all abstracted paths.

References

1. T. Ball, R. Majumdar, T. Millstein and S. Rajamani, Automatic Predicate Abstraction of C Programs, *In Proc. of PLDI 2001, SIGPLAN Notices*, Vol. 39 (2001)
2. B. Beizer, *Software Testing Techniques*, New York: Van Nostrand Reinold, (1990)
3. J.J. Chilenski and S.P. Miller, *Applicability of modified conditional coverage to software testing*, Software Engineering Journal, (1994) 193–200
4. E.M. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *In Proceedings POPL 92: Principles of Programming Languages*, (1992) 343–354
5. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for the static analysis of programs by construction of approximation of fixpoints. *In Proc. of POPL'77*, (1977) 238–252
6. D. Dams, O. Grumberg, and R. Gerth. Abstract interpretation of reactive systems: abstractions preserving ACTL*, ECTL*, and CTL*. *In Proceedings of PROCOMET 94: Programming Concepts, Methods, and Calculi*, (1994) 561–581
7. S. Graf and H. Saidi, Construction of abstract state graphs with PVS. *In Proceedings of CAV 1997*, LNCS 1254 (1997) 72–83
8. R. Hardin, Z. Har'EL, and R.P. Kurshan, COSPAN, *In Proceedings of CAV 1996*, LNCS 1102, (1996) 423–427
9. M.S. Hecht, *Flow Analysis of Computer Programs*, NY: Elsevier-N. Holland (1977)
10. J. Holzmann, *Design and Validation of Computer Protocols*, Pr. Hall, NJ (1991)
11. Kennedy Carter Inc., www.kc.com
12. Kapoor, C., and Tesar, D.: A Reusable Operational Software Architecture for Advanced Robotics (OSCAR), The University of Texas at Austin, DOE Grant No. DE-FG01 94EW37966 (1998)
13. Kurshan, R., *Computer-Aided Verification of Coordinating Processes – The Automata Theoretic Approach*, Princeton University Press, Princeton, NJ (1994)
14. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, Vol. 6(1), (1995) 11–44
15. K.S. Namjoshi and R P. Kurshan, Syntactic Program Transformations for Automatic Abstraction, *In Proc. of CAV'00*, LNCS 1855, (2000), 435–449
16. Y. Kesten and A. Pnueli, *Control and Data Abstraction: Cornerstones of the Practical Formal Verification*, Software Tools and Technology Transfer, Vol. 2(4) (2000)
17. SES Inc., *ObjectBench Technical Reference*, SES Inc. (1998)
18. N. Sharygina, *Model Checking of Software Control Systems*, Ph.D. Dissertation, The University of Texas at Austin (2002)
19. N. Sharygina, J.C. Browne and R. Kurshan, A Formal Object-Oriented Analysis for Software Reliability: Design for Verification, *In Proc. of FASE'01*, LNCS 2029, (2001), 318–332
20. S. Shlaer, S. Mellor, *Object Lifecycles: Modeling the World in States*, Pr. Hall (1992)

21. L. Starr, *Executable UML: The Models that Are the Code*, M. Integration, LLC (2001)
22. F. Xie, V. Levin, and J.C. Browne, Model Checking of an Executable Subset of UML, *In Proceedings of ASE2001: Automated Software Engineering* (2001)

Integration of Formal Datatypes within State Diagrams

Christian Attiogbé[1], Pascal Poizat[2], and Gwen Salaün[1]

[1] IRIN, Université de Nantes
2 rue de la Houssinière, B.P. 92208, 44322 Nantes Cedex 3, France
{attiogbe,salaun}@irin.univ-nantes.fr
http://www.sciences.univ-nantes.fr/info/perso/permanents/salaun/
[2] LaMI – UMR 8042 CNRS et Université d'Evry Val d'Essonne, Genopole
Tour Évry 2, 523 Place des terrasses de l'Agora, 91000 Évry, France
poizat@lami.univ-evry.fr
http://www.lami.univ-evry.fr/~poizat/

Abstract. In this paper, we present a generic approach to integrate datatypes expressed using formal specification languages within state diagrams. Our main motivations are (i) to be able to model dynamic aspects of complex systems with graphical user-friendly languages, and (ii) to be able to specify in a formal way and at a high abstraction level the datatypes pertaining to the static aspects of such systems. The dynamic aspects may be expressed using state diagrams (such as UML or SDL ones) and the static aspects may be expressed using either algebraic specifications or state oriented specifications (such as Z or B). Our approach introduces a flexible use of datatypes. It also may take into account different semantics for the state diagrams.

Keywords. formal methods integration, state diagrams, algebraic specifications, Z, B

1 Introduction

The joint use of formal and semi-formal specification languages is a promising approach, with the objective of taking advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. In this paper, we propose an approach dealing with this issue. It enables one to specify the different aspects of complex systems using an integrated language.

Static and functional aspects are specified using static formal specification languages (algebraic specifications [6], state oriented languages such as Z [24] or B [1]). This makes the verification of specifications possible but also the datatypes description at a very high abstraction level. The flexibility we propose at the static aspects specification level enables the specifier to choose the formal language that is the more suited to this task: either the one (s)he knows well, the one with tools, or the one that makes possible the reuse of earlier specifications.

Dynamic aspects (*i.e.* behaviour, concurrency, communication) are modelled using state diagrams. Our proposal is generic. Different dynamic semantics may be taken into account, hence our approach may be used for Statecharts [17], for the different

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 341–355, 2003.

(yet growing number) of UML state diagrams semantics, [21,20,25,19] for instance, and more generally for any state / transition oriented specification. In our approach, the specification is control-driven: the dynamic aspects are the main aspects within a specification and state how the static aspects datatypes are used. On a larger scale, our work deals with the formal specifications integration and composition issues, where we yet have some general results [3]. At the global specification level, our approach therefore also addresses the consistency of the static and dynamic parts.

This paper is structured as follows. In Sect. 2, we present the syntactic extensions used to integrate formal datatypes within state diagrams. Section 3 deals with the generic integration semantics. To end, Sect. 4 concludes the paper and presents some perspectives.

2 Syntax

We here present the syntactic extensions we add to state diagrams to take into account the formal datatypes integration. We advocate for a control-driven approach of mixed specification. This means that the main part of a specification is given by the dynamic aspects modelling (behaviours and communications). The static aspects are encapsulated within the dynamic aspects.

We first add module[1] importation and local variables declaration extensions to state diagrams. Both are done using *data boxes* (Fig. 1), a kind of UML-inspired note which is usually used to give additional informations to a diagram in a textual form. The `IMPORT` notation is introduced to indicate which data modules are imported as well as the language used to write their contents (algebraic specifications, Z schemas, B machines, or other). Such a language is called a *framework* in our approach and is used to define the specific evaluation functions which enable us to evaluate the data embeddings into state diagrams. Variables are also declared and typed in the data boxes. Since modules often contain several type definitions, and since types with the same name may be defined in two different modules, the type of a variable may be prefixed with the name of a module in order to avoid conflicts.

The general form of a state diagram transition is made up of two states (which can contain activities) and a label with an event, a guard and an action to do when

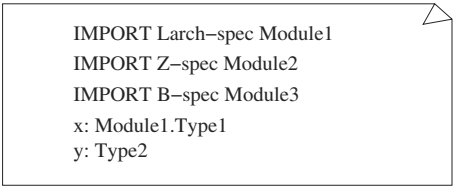


Fig. 1. Local declarations of data into state diagrams

[1] A module (general definition) is a collection of one or several descriptions of datatypes or behaviours.

Table 1. Links between state diagrams and data

Part of label	Kind of interaction	Example with data
EVENT	reception	*event-name*(x_1:T_1, ... ,x_n:T_n)
GUARD	guard	*predicate*(t_1, ... ,t_n)
ACTION	emission	*receiver* ^ *event-name*(t_1, ... ,t_n)
ACTION	local modification	x:=t

the transition is triggered. Datatype extensions (*data expressions*) may appear in both states and transitions. In states, our extensions correspond to activities (actions such as entry/exit or local modifications, and events). In transitions, our extensions (Table 1) enable one to (i) receive values in events and then store these values into local variables, (ii) guard transitions with data expressions, (iii) send events containing data expressions and (iv) make assignments of data expressions to the local variables. The last two take place in the action part of transitions. Possible extended activities within states are not directly taken into consideration in our approach. However, such activities can be viewed as a specific case of transitions between states.

Data expressions may be either variables, terms for algebraic specifications or operation applications for state oriented specifications. As far as the formal language for the static aspects is concerned, the only constraint is to have some well-defined evaluation mechanism. The reason for this is that we are interested in operational semantics to address in a generic way operational issues of mixed specifications (animation, equivalences/bisimulations, model checking) which have been addressed for specific mixed specification languages (*e.g.* LOTOS [11]). Denotational issues may be treated with denotational approaches based on institutions for example [16,2] and would enable to ease this evaluation mechanism constraint. Our approach makes possible the joint use of several static formal languages at the same time. However, a strong mix of constructs from several languages (such as importing a Z module within an algebraic specification, or using algebraic specification variables in a Z operation application) is prohibited in order to avoid possible semantic inconsistencies. Our goal is neither to advocate for such complex combinations, nor to develop a mean to solve such inconsistencies. As a simple way to detect them, we develop a *meta-type* concept using meta-typing rules (see the semantics below). Terms which are not meta-typed (*i.e.* inconsistent) will not be able to be used in the dynamic rules.

3 Semantics

In this section, our goal is to give a formal semantics to state diagrams extended with formal datatypes in a way that has been presented in the syntactic part. We do not aim at formalizing some specific kind of state diagram, which has already successfully been done, see [21,20,25,19,17] for example. We rather aim at being able to reuse different existing state diagram semantics. Therefore, our semantics is presented in such a way that generic concepts may then be instantiated for a specific kind of state diagram. In our semantics we will deal with properties such as “the event pertains to the input event collection of the state diagram”, which could thereafter be instantiated for a specific

state diagram language into "the event is the first element of the input queue associated with the state diagram process". Such generic concepts are represented in our semantics using boxes (*e.g.* $event \boxed{\in} \mathcal{Q}_{in}$).

Using this generic approach, we preserve a very general description of extended state diagrams. In this way, all kinds of state diagrams and their underlying semantics could be considered. The only constraint is that the semantics of a non-extended state diagram D has to be given in terms of a Labelled Transition System (LTS) ($\boxed{INIT}(D)$, $\boxed{STATE}(D)$, $\boxed{TRANS}(D)$), which is currently always the case. We may then define precisely the meaning of the integration of data into dynamic diagrams using *extended states* evolutions and *evaluation functions*.

Our semantics is given using rules which may be decomposed into: meta-typing rules (badly meta-typed terms may not be evaluated), action evaluation rules (describing the effects of actions on extended states), dynamic rules (formalizing the individual evolution of an extended state diagram), open systems rules (describing the effect of putting extended state diagrams within an external environment), and global system and communication rules (putting things altogether). We will also discuss the evaluation functions associated with the different static specification languages one may use. To end this section on the semantics of our approach, we will illustrate the application of our semantic rules on a simple piece of specification. Specific notations are introduced throughout the semantic definitions.

Meta-Typing Rules. These rules are needed in order to detect multiple language inconsistencies and to be able to perform the evaluation of a term using the adequate evaluation function, that is the one dedicated to its meta-type (*e.g.* Larch, CASL, Z, B). In the following, $\mathcal{D}$ corresponds to the set of extended state diagrams. Within the rules we are in the context of a specific diagram D pertaining to $\mathcal{D}$. The states of D are denoted by $STATE(D)$, its initial states by $INIT(D)$ (which is a subset of $STATE(D)$), and its transitions by $TRANS(D)$. $DeclImp(D)$ and $DeclVar^!(D)$ denote respectively the imports and the variables declarations of the diagram D data box. $DeclVar^?(D)$ is the set of (typed) variables received in events. $DeclVar$ is the union of $DeclVar^?(D)$ and $DeclVar^!(D)$. A diagram D may be given syntactically by a tuple ($INIT$, $STATE$, $TRANS$, $DeclImp$, $DeclVar^!$). $def(x^{\cdot} M)$ is true if x is defined within the module M. We use T for usual types and X for meta-types. The notation $t ::_D X$ means that t has X for meta-type within the diagram D. Throughout the semantic part, operators suffixed with meta-types (*e.g.* $\rhd_X$) will denote their interpretation within the context of the corresponding framework (*e.g.* $\rhd_B$ denotes the B evaluation function). The meta-typing rules are given in Fig. 2.

$$\frac{\texttt{IMPORT}\ X-SPEC\ M \in DeclImp(D) \quad def(T, M) \quad x : T \in DeclVar(D)}{x ::_D X}\ (a) \qquad \frac{\texttt{IMPORT}\ X-SPEC\ M \in DeclImp(D) \quad def(op, M) \quad \forall\, i \in 1..n\ .\ t_i ::_D X}{op\ \ t_1\ \ldots\ t_n ::_D X}\ (b)$$

Fig. 2. Meta-typing rules

The Fig. 2a rule is used to give a meta-type to variables using data local declarations. The Fig. 2b rule gives the meta-type of a construction from the meta-types of elements which compose it. $op \quad t_1 \ \triangleright\!\triangleright\!\triangleright \ t_n$ is an abstract notation to denote the application of an operation to a list of terms, since syntactically there are some differences between algebraic and state oriented formal specification languages.

Action Evaluation Rules. This set of rules deals with the effect of actions on the extended states used to give semantics to extended state diagrams. Let us first give a definition of these states. $EVENT^?$ is the set of all *input events*, whose general form is $event-name(value_1 \lhd\!\rhd\!\rhd value_n)$, that is a concrete instantiation with values of an abstract event parameterized by variables (*e.g.* $e(0)$ is an instantiation of $e(x : Nat)$). $EVENT^!$ is the set of all *output events*, whose general form is $receiver \hat{} \ event-name(value_1 \lhd\!\rhd\!\rhd value_n)$. $EVENT$ is the set of all events, that is: $EVENT = EVENT^? \cup EVENT^!$. The set of extended states for an extended state diagrams is defined as:

$$\mathcal{S} \subseteq \boxed{STATE}(D) \times \mathcal{E} \times \boxed{\mathcal{Q}}[EVENT^?] \times \boxed{\mathcal{Q}}[EVENT^!]$$

where

- $\boxed{STATE}(D)$ is the set of states used to give a semantics to the non-extended state diagram D;
- $\mathcal{E}$ is the set of environments, which are finite sets of pairs $(x \lhd v)$ denoting that the variable x is bound to the value v;
- $\boxed{\mathcal{Q}}$ is the set of collections[2], $\boxed{\mathcal{Q}}[EVENT^?]$ the set of input events collections, and $\boxed{\mathcal{Q}}[EVENT^!]$ the set of output events collections.

Collections are introduced to memorize events exchanged between diagrams. Q_{in_D} (resp. Q_{out_D}) is used to denote a collection associated to diagram D to store input (resp. output) events. The $E \vdash e \triangleright_X v$ notation means that using the evaluation defined in the X framework, v is a possible evaluation of e using the environment E for substituting the free variables. More details concerning the semantics of $\triangleright_X$ will be given in the remainder. Furthermore, if E and E' are environments then EE' is the environment in which variables of E and E' are defined and the bindings of E' overload those of E. We recall that symbols in boxes depict abstract structures and operations to be instantiated for a given type of state diagram. S will thereafter be used to denote an element of $\mathcal{S}$, and $\square_D$ to denote an element of $\boxed{STATE}(D)$. When there is no ambiguity on D, we use $\square$ for $\square_D$, Q_{in} for Q_{in_D}, and Q_{out} for Q_{out_D}. The rules describing the evaluation of actions are given in Fig. 3.

The $EVAL-SEQ$ rule is used to evaluate the actions in sequence. $a_1 \lhd\!\rhd\!\rhd a_n$ is an abstract notation for a sequence of actions as we do not wish to set here some particular concrete syntax for these. The $EVAL-NIL$ rule states that doing no action does not change the global state. The event emissions are dealt with by the $EVAL-SEND$ rule,

[2] A collection is an abstract structure which may be instantiated for a given type of state diagram by a queue, a set or a multiset for example.

$$\frac{act-eval(a_1, S, D) = S' \quad act-eval(a_2 \ldots a_n, S', D) = S''}{act-eval(a_1 \ldots a_n, S, D) = S''}\ EVAL\text{–}SEQ \qquad \frac{}{act-eval(\varepsilon_{act}, S, D) = S}\ EVAL-NIL$$

$$\frac{\forall\, i \in 1..n\ .\ \exists\, X_i\ .\ t_i ::_D X_i\ .\ \exists\, v_i\ .\ E \vdash t_i \rhd_{X_i} v_i}{act-eval(rec\hat{}e(t_1, \ldots, t_n), < \Gamma, E, Q_{in}, Q_{out} >, D) = < \Gamma, E, Q_{in}, Q_{out} \boxed{\uplus} \{rec\hat{}e(v_1, \ldots, v_n)\} >}\ EVAL-SEND$$

$$\frac{\exists\, X\ .\ t ::_D X\ .\ \exists\, v\ .\ E \vdash t \rhd_X v}{act-eval(x := t, < \Gamma, E, Q_{in}, Q_{out} >, D) = < \Gamma, E\{x \leftarrowtail v\}, Q_{in}, Q_{out} >}\ EVAL\text{–}ASSIGN$$

Fig. 3. Action evaluation rules

which expresses that the effect of sending an event is to evaluate its arguments and then put it[3] into the state diagram output event collection.

The $EVAL - ASSIGN$ rule may need some explanations. Roughly speaking, assignments update the local environment. The understanding of this rule in an algebraic specification framework is quite natural (v is an interpretation of t). For state oriented languages, the interpretation of the rule is slightly different. x is a state variable. The value v denotes the new state obtained from the environment and after the evaluation of t. The notation used in this rule is specific to our approach and slightly different of the usual notation (*i.e.* pointed or with side effect). However, we wish to have at our disposal a common notation for the different considered languages.

Dynamic Rules. This set of rules deals with the dynamic evolution of a single state diagram. We introduce a special event, $\square$ denoting (as usual) a stuttering step.

$$EVENT^{?+} = EVENT^{?} \cup \{\square\}$$

The state diagram evolutions are given in terms of a LTS ($\underline{INIT}$, $\underline{STATE}$, $\underline{TRANS}$) where states are extended states, with:

$$\underline{STATE} \subseteq \mathcal{S}$$
$$\underline{INIT} \subseteq \underline{STATE}$$
$$\underline{TRANS} \subseteq \underline{STATE} \times EVENT^{?+} \times \underline{STATE}$$

We recall that extended state diagrams may be given syntactically as a tuple ($INIT$, $STATE$, $TRANS$, $DeclImp$, $DeclVar^{!}$) and that the semantics of non-extended state diagrams are given in terms of a LTS ($\boxed{INIT}(D)$, $\boxed{STATE}(D)$, $\boxed{TRANS}(D)$). For some kind of state diagrams, there is a correspondence of the notion of states (*i.e.* $INIT$, $STATE$, $\boxed{INIT}$ and $\boxed{STATE}$ have the same type). However, for the others (such as the UML state diagrams), the semantics of non-extended state diagrams are given in

[3] $\boxed{\uplus}$ denotes an abstract union operation which may be instantiated differently depending on the type of state diagram semantics we want: union, put in front of a queue, etc.

terms of *configurations* [21,20,25] which are sets of *active* states. The $active$ function is used to know if a state is active in a configuration (or more generally in some element of $\boxed{STATE}$). $active(\square, \square)$ yields true if $\square$ is active in $\square$.

To be able to reuse the semantic information yield by $\boxed{TRANS}(D)$, we have first to define a function that maps members of $TRANS(D)$ (extended transitions) to members of $\boxed{TRANS}(D)$. This function, $base$, is defined inductively on the structure of the extended diagram notation [8]. A first rule (Fig. 4) is used to obtain the initial extended states which correspond to the initial states of the non-extended state diagram underlying semantics ($\boxed{INIT}(D)$) extended with initial values for the variables and empty input and output collections ($\boxed{\emptyset}$).

$$\frac{\begin{array}{c}\exists\, \gamma_0 \in INIT(D)\ .\\ \exists\, \Gamma_0 \in \boxed{INIT}(D)\ .\\ active(\gamma_0, \Gamma_0)\\ DeclVar^{!}(D) = \cup_{i\in 1..n}\{x_i : T_i\}\\ \forall\, i \in 1..n\ .\ \exists\, X_i\ .\ x_i ::_D X_i\ .\ \exists\, v_i :_{X_i} T_i\end{array}}{< \Gamma_0, \cup_{i\in 1..n}\{x_i \mapsto v_i\}, \boxed{\emptyset}, \boxed{\emptyset} > \in \underline{INIT}(D)}\ DYN-INIT$$

Fig. 4. Initialization rule

The meta-type of variables is used to define the notion of type in terms of a specific framework, which is noted $v :_X T$. Hence, the notation $\exists\, v :_X T$ denotes the fact that, within the X framework, v is a value of type T. A second rule (Fig. 5) is used to express stuttering steps. These steps denote an extended state diagram doing no evolution and will be used when putting state diagrams in an open system environment.

$$\frac{S \in \underline{STATE}(D)}{S \xrightarrow{\varepsilon} S \in \underline{TRANS}(D)}\ DYN-\varepsilon$$

Fig. 5. Stuttering steps rule

The next dynamic rule (Fig. 6) expresses the general evolution triggered when an event is read from the state diagram input event collection. This event may carry data values that are put into the state diagram variables environment. $TRUE_X$ denotes the truth value within the X framework.

However, sometimes there are no events to trigger transitions. Such a case is dealt with by Fig. 7 rule, where the corresponding transition of $\underline{TRANS}(D)$ is labelled by the stuttering step label ($\square$).

Once again, boxed elements are abstract concepts to be instantiated for a given type of state diagram semantics. $e \boxed{\in} \mathcal{Q}$ denotes that the event e is in the collection $\mathcal{Q}$. Possible instantiations are: e is in $\mathcal{Q}$ ($e \in \mathcal{Q}$), e is the first/top element in $\mathcal{Q}$ ($e = car(\mathcal{Q})$), e is

$$\frac{\begin{array}{c} S \in \underline{STATE}(D) \\ S =< \Gamma, E, Q_{in}, Q_{out} > \\ \exists\, \gamma \in STATE(D)\ .\ \exists\, \gamma' \in STATE(D)\ . \\ \exists\, l = e(x_1 : T_1, \ldots, x_n : T_n)\ g\ /\ a_1 \ldots a_m\ . \\ \gamma \xrightarrow{l} \gamma' \in TRANS(D) \\ active(\gamma, \Gamma) \\ \exists\, e(v_1, \ldots, v_n) \boxed{\in} Q_{in} \\ Q'_{in} = Q_{in} \boxed{\setminus} \{e(v_1, \ldots, v_n)\} \\ \exists\, \Gamma \xrightarrow{base(l)} \Gamma' \in \boxed{TRANS}(D) \\ \forall\, i \in 1..n\ .\ \exists\, X_i\ .\ x_i ::_D X_i\ .\ \exists\, v_i :_{X_i} T_i \\ E' = E \cup_{i \in 1..n} \{x_i \mapsto v_i\} \\ \exists\, X\ .\ g ::_D X \\ E' \vdash g \rhd_X TRUE_X \\ act-eval(a_1 \ldots a_m, < \Gamma, E', Q'_{in}, Q_{out} >, D) =< \Gamma, E'', Q'_{in}, Q'_{out} > \\ S' =< \Gamma', E'', Q'_{in}, Q'_{out} > \end{array}}{\begin{array}{c} S' \in \underline{STATE}(D) \\ S \xrightarrow{e(v_1, \ldots, v_n)} S' \in \underline{TRANS}(D) \end{array}}\ DYN-E$$

Fig. 6. Basic dynamic rule (event reception)

$$\frac{\begin{array}{c} S \in \underline{STATE}(D) \\ S =< \Gamma, E, Q_{in}, Q_{out} > \\ \exists\, \gamma \in STATE(D)\ .\ \exists\, \gamma' \in STATE(D)\ . \\ \exists\, l = g\ /\ a_1 \ldots a_m\ . \\ \gamma \xrightarrow{l} \gamma' \in TRANS(D) \\ active(\gamma, \Gamma) \\ \exists\, \Gamma \xrightarrow{base(l)} \Gamma' \in \boxed{TRANS}(D) \\ \exists\, X\ .\ g ::_D X \\ E \vdash g \rhd_X TRUE_X \\ act-eval(a_1 \ldots a_m, < \Gamma, E, Q_{in}, Q_{out} >, D) =< \Gamma, E', Q'_{in}, Q'_{out} > \\ S' =< \Gamma', E', Q'_{in}, Q'_{out} > \end{array}}{\begin{array}{c} S' \in \underline{STATE}(D) \\ S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(D) \end{array}}\ DYN-E\emptyset$$

Fig. 7. Basic dynamic rule (no event reception)

the element in $\mathcal{Q}$ with the highest priority, and so on. $e \boxed{\setminus} \mathcal{Q}$ denotes, in the same way, the (abstract) removal of e out of $\mathcal{Q}$. $\boxed{TRANS}(D)$ is the set of transitions of the non-extended state diagram semantics. The $DYN-E$ and $DYN-E\emptyset$ rules deal with the more general forms of state diagrams transitions (*i.e.* with the EVENT [GUARD]/ACTION and [GUARD]/ACTION forms). Rules for restricted forms of transitions (*e.g.* without guard) may be obtained in an easy way from these general rules (*e.g.* consider the guard

to be true). An operational semantics is easily obtained from this model associating to D its LTS ($\underline{INIT}(D)$, $\underline{STATE}(D)$, $\underline{TRANS}(D)$), and then using an usual trace semantics (TR) for example:

$$|| \ D \ ||_{oper} = \ TR(\underline{INIT}(D)^\frown \underline{STATE}(D)^\frown \underline{TRANS}(D))$$

Open System Rules. This set of rules is used to express what happens when a single state diagram is put into an open system environment. Mainly, the intuition we want to express is that some events may be received from the environment and some other may be send to it. As far as the input and output event collections of the state diagrams are concerned, this means that things are put into the input collection and things are taken out of the output collection. Something important to note is that these modifications of the extended states may appear meanwhile the state diagram does a transition (*i.e.* following the $DYN-E$ and $DYN-E\emptyset$ rules) but also if it does nothing. To be able to represent this, we may use the $\square$ transitions (rule $DYN-\square$). Let us now formalize this intuition. As usual in our approach, the semantics of a state diagram in an open system will be defined as the traces of the LTS ($\underline{INIT}^{open}(D)^\frown \underline{STATE}^{open}(D)^\frown \underline{TRANS}^{open}(D)$), that is:

$$|| \ D \ ||^{open}_{oper} = \ TR(\underline{INIT}^{open}(D)^\frown \underline{STATE}^{open}(D)^\frown \underline{TRANS}^{open}(D))$$

For a given diagram D, we state that:

$$\underline{INIT}^{open}(D) \subseteq \underline{INIT}(D)$$
$$\underline{TRANS}^{open}(D) \subseteq \underline{TRANS}(D) \times \boxed{\mathcal{Q}}[EVENT^?] \times \boxed{\mathcal{Q}}[EVENT^!]$$
$$\underline{STATE}^{open}(D) \subseteq SOURCE(\underline{TRANS}^{open}(D)) \cup TARGET(\underline{TRANS}^{open}(D))$$

with functions $SOURCE$ and $TARGET$ respectively denoting the source and target extended states of transitions. A unique rule, $DYN-OPEN$ (Fig. 8), is then defined to express the collection modifications that may take place in an open system semantics. In this rule, the label l matches the two possible things the state diagram may do during the collections modification: nothing (rule $DYN-\square$) or a classical transition (rules $DYN-E$ and $DYN-E\emptyset$).

$\boxed{\mathcal{P}}(E)$ denotes the collection obtained from the powerset of a set E. E_{in} (resp. E_{out}) in open transitions (members of $\underline{TRANS}^{open}(D)$) is used to keep the information of what has been put into the input event collection (resp. taken out of the output event collection) of the state diagram. $S \xrightarrow{l}_{E_{in}{}^\frown E_{out}} S' \in \underline{TRANS}^{open}(D)$ is used as a shorthand notation for $(S^\frown l^\frown S'^\frown E_{in}{}^\frown E_{out}) \in \underline{TRANS}^{open}(D)$.

Global System and Communication. We may now define the last set of rules, putting things altogether. The idea is that a global system made up of several extended state diagrams (denoted $\cup_{i \in 1 \bowtie n} D_i$) will evolve as its component evolve. Once again we will define the operational semantics of the system to be the traces of a LTS ($\overline{INIT}(\cup_{i \in 1 \bowtie n} D_i)$, $\overline{STATE}(\cup_{i \in 1 \bowtie n} D_i)$, $\overline{TRANS}(\cup_{i \in 1 \bowtie n} D_i)$), that is:

$$|| \cup_{i \in 1 \bowtie n} D_i \ ||^{open}_{oper} =$$
$$TR(\overline{INIT}(\cup_{i \in 1 \bowtie n} D_i)^\frown \overline{STATE}(\cup_{i \in 1 \bowtie n} D_i)^\frown \overline{TRANS}(\cup_{i \in 1 \bowtie n} D_i))$$

$$\frac{\begin{array}{c}< \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l} < \Gamma', E', Q'_{in}, Q'_{out} > \in \underline{TRANS}(D) \\ \exists\, E_{out} \subseteq Q_{out} \\ \exists\, E_{in} \subseteq \boxed{\mathcal{P}}(EVENT^{?})\end{array}}{< \Gamma, E, Q_{in}, Q_{out} > \xrightarrow{l}_{E_{in},E_{out}} < \Gamma', E', Q'_{in} \boxed{\uplus} E_{in}, Q'_{out} \boxed{\setminus} E_{out} > \in \underline{TRANS}^{open}(D)}\ DYN\text{–}OPEN$$

Fig. 8. Open dynamic rule

We now have to define $\overline{INIT}$, $\overline{STATE}$ and $\overline{TRANS}$.

$$\overline{INIT}(\cup_{i\in 1\bowtie n} D_i) \subseteq \Box_{\,i}\, \underline{INIT}^{open}(D_i)$$
$$\overline{TRANS}(\cup_{i\in 1\bowtie n} D_i) \subseteq \{t \in \Box_{\,i}\, \underline{TRANS}^{open}(D_i) \mid CC(t)\}$$
$$\overline{STATE}(\cup_{i\in 1\bowtie n} D_i) \subseteq \overline{INIT}(\cup_{i\in 1\bowtie n} D_i) \cup TARGET(\overline{TRANS}(\cup_{i\in 1\bowtie n} D_i))$$

$\overline{STATE}$ is obtained from initial states ($INIT$) and reachable states (target states of transitions). $\overline{TRANS}$ is obtained from the product of the $\underline{TRANS}^{open}$ sets of each state diagram of the system, restricting this product with a communication constraint (CC) which expresses that whenever an emission event is taken out of a given diagram (D_k) output event collection (*i.e.* present in E_{out_k}), and if the receiver of this emission is a member (D_j) of the system, then this receiver has the event being put into its input event collection (E_{in_j}).

$$CC(S_1 \xrightarrow{l_1}_{E_{in_1} \lhd E_{out_1}} S'_1 \lhd\rhd\rhd \lhd S_n \xrightarrow{l_n}_{E_{in_n} \lhd E_{out_n}} S'_n) \Leftrightarrow$$
$$\forall\, k \in 1\bowtie n \rhd \forall\, D_j \hat{}\, e \boxed{\in} E_{out_k} \rhd D_j \in \cup_{i\in 1\bowtie n} D_i \Longrightarrow e \boxed{\in} E_{in_j}$$

Other specific communication contraints may be defined in order to take different communication semantics into account.

Semantics of $\rhd_X$. Several kinds of evaluation functions may be defined depending on which data specification language is used. In this paper, we focus on state oriented languages and especially on Z. Comprehensive explanations concerning algebraic specifications and B are reported in [8].

Z [24] is a mathematical notation based on set theory and first order predicate calculus. Z defines schemas to structure data specifications and operation specifications. A schema is made up of a declaration part (a set of typed schema variables) and a predicate part built on these variables. The semantics of a state schema consists in a set of bindings between the schema variables and values such that the predicate holds. State schemas define state spaces. A complete Z specification also uses an initialization schema which defines initial values for the variables.

The idea to define the Z evaluation function is to consider LTSs associated with Z specifications. This is possible since Z follows a model oriented approach. For this purpose, we use both the state schema, the initialization schema and the operation schemas of Z specifications. Let z be a Z specification defined with a state schema $SSch_z$, an initialization schema $SInit_z$, and a set of operation schemas. We may then define the associated LTS. Its set of states ($STATE_z$) corresponds to the $SSch_z$ semantics state space. The set of initial states of the LTS is the subset of $STATE_z$ with elements that satisfy the

predicate of $SInit_z$. Finally, each operation schema predicate, used to relate the bindings of two states, defines a set of transitions labelled by operation applications. The set of transitions of the LTS ($TRANS_z$) is the union of all these transitions. The evaluation function $\triangleright_Z$ is then defined as: $E \vdash l \triangleright_Z s' \Leftrightarrow \exists\, s \subseteq E \triangleright s \xrightarrow{l} s' \in TRANS_z$.

Application. In this part we illustrate the application of our semantic rules on a simple example with three communicating state diagrams (Fig. 9).

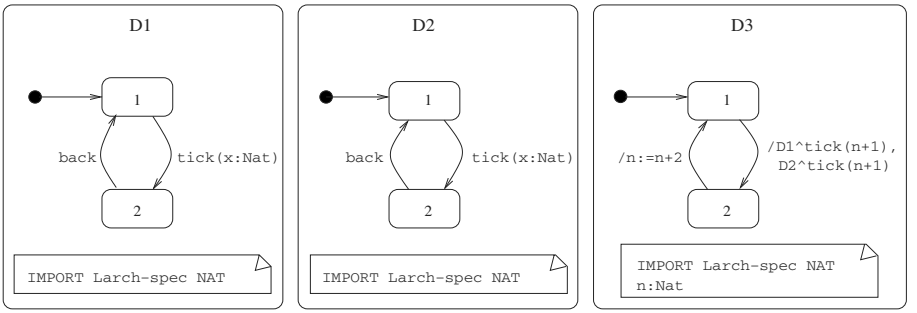


Fig. 9. A simple data extended state diagrams system

We first have to choose a non-extended state diagram semantics, like for example the semantics of UML state diagrams given by Jürjens [19]. $\boxed{INIT}$, $\boxed{STATE}$ and $\boxed{TRANS}$, used to define a semantics to non-extended state diagrams, are instantiated from [19]. $\boxed{INIT}(D)$, for example, correspond to the initial states computed using the **SCInitialize(D)** rule formalized in [19]. However, in this example we will still use the abstract notations to keep the rules readable. Each generic concept within our rules ($\boxed{\emptyset}$, $\boxed{\mathcal{Q}}$, $\boxed{\in}$, $\boxed{\setminus}$, $\boxed{\uplus}$) is instantiated by some concrete concept (resp. nil, Queue, car, cdr, append). Here, the event collections are queues, together with the usual operations on them.

Our example uses natural numbers (sort `Nat`) defined in an usual way using `0` and `succ` [8]. `Nat` is written using the input language of the LP theorem prover [15], which is a subset of Larch. Within this framework, the evaluation function corresponds to term rewriting, *i.e.* $\triangleright_{Larch\text{–}spec} \equiv \leadsto^*_R$ with R being the set of rewrite rules. The rewrite rules are obtained from the algebraic specification axioms applying the **noeq-dsmpos** LP ordering command [15]. $TRUE_{Larch\text{–}spec}$ corresponds to $true$ in LP.

As a first example of rule application, Fig. 10 gives an example of a simple dynamic evolution. This is an instantiation of the $DYN - E\emptyset$ rule (Fig. 7) without guard. It represents an independent evolution of diagram D3 from state 2 to state 1. The `w` value bound to the `n` variable is supposed to be the normal form of the `n+2` term. This evaluation is performed through the $act\text{–}eval$ application, and the underlying semantic rule $EVAL\text{–}SEND$ of Fig. 3 (having as premise: $\{(\mathtt{n} \cdot \mathtt{v})\} \vdash \mathtt{n} + 2 \leadsto^*_R \mathtt{w}$). The conclusion of the rule denotes the state and transition construction of the diagram model.

$$\frac{\begin{array}{c} S \in \underline{STATE}(\mathrm{D3}) \\ S =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, \emptyset, nil > \\ l = / \ \mathtt{n} := \mathtt{n} + \mathtt{2} \\ \gamma_3 \xrightarrow{l} \gamma_3' \in TRANS(\mathrm{D3}) \\ active(\gamma_3, \Gamma_3) \\ \Gamma_3 \xrightarrow{\varepsilon_{act}} \Gamma_3' \in \boxed{TRANS}(\mathrm{D3}) \\ act-eval(\mathtt{n} := \mathtt{n} + \mathtt{2}, < \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, nil >, \mathrm{D3}) =< \Gamma_3, \{(\mathtt{n},\mathtt{w})\}, nil, nil > \\ S' =< \Gamma_3', \{(\mathtt{n},\mathtt{w})\}, nil, nil > \end{array}}{\begin{array}{c} S' \in \underline{STATE}(\mathrm{D3}) \\ S \xrightarrow{\varepsilon} S' \in \underline{TRANS}(\mathrm{D3}) \end{array}}$$

Fig. 10. Independent dynamic evolution

An example of communication is given in Figs. 11, 12, and 13. It describes the asynchronous communication on `tick` between D1, D2 and D3 (which sends the `tick` events). In this example, we assume that the diagrams D1 and D2 are initially in state 1 and that the diagram D3 is in state 2 with the two sent events in its output queue. Figure 11 describes a conjunction of evolutions for the three diagrams where D3 has events taken out of its output queue whereas D1 and D2 have events put into their input queues. Such individual evolutions (here in premises) could have been proven correct, independently for each diagram, using dynamic evolution rules (such as the one given in Fig. 10) and then the open system rule (expressing the possible queues evolutions). We do not give the whole proof here by lack of place. Figure 12 states that the communication constraint is verified for Fig. 11 global evolution and therefore this evolution is a legal evolution for the global system (Fig. 13). Note that in the rules, the abstract concepts have been instantiated by concrete concepts, with $queue$ being the $Queue$ constructor.

$$\frac{\begin{array}{c} S_1 =< \Gamma_1, \emptyset, nil, nil > \quad \wedge \quad S_1' =< \Gamma_1, \emptyset, queue(\mathtt{tick(u)}), nil > \\ T_1 = S_1 \xrightarrow{\varepsilon}_{queue(\mathtt{tick(u)}), nil} S_1' \\ S_2 =< \Gamma_2, \emptyset, nil, nil > \quad \wedge \quad S_2' =< \Gamma_2, \emptyset, queue(\mathtt{tick(u)}), nil > \\ T_2 = S_2 \xrightarrow{\varepsilon}_{queue(\mathtt{tick(u)}), nil} S_2' \\ S_3 =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, queue(\mathrm{D1}\,\hat{}\,\mathtt{tick(u)}, \mathrm{D2}\,\hat{}\,\mathtt{tick(u)}) > \\ S_3' =< \Gamma_3, \{(\mathtt{n},\mathtt{v})\}, nil, nil > \\ T_3 = S_3 \xrightarrow{\varepsilon}_{nil, queue(\mathrm{D1}\,\hat{}\,\mathtt{tick(u)}, \mathrm{D2}\,\hat{}\,\mathtt{tick(u)})} S_3' \\ T = (T_1, T_2, T_3) \end{array}}{T \in \Pi_{i \in 1..3} \underline{TRANS}^{open}(D_i)}$$

Fig. 11. Events exchange

4 Discussion

Our goal is to propose specifier-friendly integrated languages for mixed specifications, and more generally an integration method suited to the specification of mixed complex

$$CC(S_1 \xrightarrow{\varepsilon}_{queue(\texttt{tick(u)}),nil} S'_1, S_2 \xrightarrow{\varepsilon}_{queue(\texttt{tick(u)}),nil} S'_2, S_3 \xrightarrow{\varepsilon}_{nil,queue(\texttt{D1\^{}tick(u)},\texttt{D2\^{}tick(u)})} S'_3) \Longleftrightarrow$$
$$(\texttt{D1\^{}tick(u)} = car(queue(\texttt{D1\^{}tick(u)}, \texttt{D2\^{}tick(u)})) \quad \wedge \quad \mathrm{D_1} \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow$$
$$\texttt{tick(u)} = car(queue(\texttt{tick(u)}))) \quad \wedge$$
$$(\texttt{D2\^{}tick(u)} = car(queue(\texttt{D1\^{}tick(u)}, \texttt{D2\^{}tick(u)})) \quad \wedge \quad \mathrm{D_2} \in \cup_{i\in 1..3}\mathrm{D_i} \Longrightarrow$$
$$\texttt{tick(u)} = car(queue(\texttt{tick(u)})))$$

Fig. 12. Instantiation of the CC rule

$$\frac{T = (S_1 \xrightarrow{\varepsilon}_{queue(\texttt{tick(u)}),nil} S'_1, S_2 \xrightarrow{\varepsilon}_{queue(\texttt{tick(u)}),nil} S'_2, S_3 \xrightarrow{\varepsilon}_{nil,queue(\texttt{D1\^{}tick(u)},\texttt{D2\^{}tick(u)})} S'_3) \quad T \in \Pi_{i\in 1..3}\overline{TRANS}^{open}(D_i) \mid CC(T)}{T \in \overline{TRANS}(\cup_{i\in 1..3}\mathrm{D_i})}$$

Fig. 13. Example of transition in the final semantic model

systems. We choose to combine state diagrams with formal specification languages devoted to abstract datatypes (algebraic specifications or state oriented specifications such as Z and B). This joint use, in a formal and integrated framework, of a semi-formal notation for dynamic aspects with formal languages for static aspects enables one to take advantage of both approaches: specifier-friendliness and readability from semi-formal approaches, high abstraction level, expressiveness, consistency and verification means from formal approaches. For an example of a case study specified using our approach, the reader may refer to [9].

Comparison. As a complement to the description of datatypes using class diagrams, the usual extension of state diagrams is the use of OCL (Object Constraint Language) constraints. However, OCL is not really suited to the description of formal abstract datatypes. There are also numerous works combining state diagrams with Z or B such as [10,23]. In both cases, authors proceed using a translation into the static formalism (Z or B) which thereafter constitutes an homogeneous framework for the subsequent steps of the formal development. Casl-Chart [22] combines Statecharts (following the STATEMATE semantics [17]) with the CASL algebraic specification language [4]. A Casl-Chart specification is made up of datatypes written in CASL and several Statecharts that may use these datatypes in events, guards and actions. Astesiano *et al.* [5] suggest a method to compose languages, in particular a data description language and a paradigm-specific language. Their goal is the description of languages in a component-based style, focusing on the data definition component. Unlike all these existing approaches, our proposal enables to take into account the different UML state diagrams semantics, and more generally Statecharts semantics or any states and transitions formalisms such as the SDL [14] or the recent works on symbolic transition systems [18,11,12]. Our approach is also more flexible as it permits to use different datatype description languages, hence increasing the reusability level of specifications.

Perspectives. A first perspective addresses verification issues. If it is yet possible to verify the aspects taken separately, it is important to be able to verify the global system. We are working on the translation of our generic approach for integrated mixed languages and its semantics into higher order logic tools such as PVS [13]. We also

have developed a tool dedicated to the animation of specifications combining CCS with abstract datatypes. This tool, ISA [7], is quite generic over the datatype (*i.e.* static) language which is concerned. The next step will then be to extend ISA in order to deal with several dynamic specification languages. Finally, a generalization of our approach represents an interesting challenge in order to be able to combine different formalisms based on the integration of formal datatypes within state / transition systems (such as SDL [14]).

References

1. J.R. Abrial. *The B-Book*. Cambridge University Press, 1996.
2. M. Aiguier, F. Barbier, and P. Poizat. A Logic for Mixed Specifications. Technical Report 73-2002, LaMI, Germany, 2002. Presented at WADT'2002.
3. M. Allemand, C. Attiogbé, P. Poizat, J.-C. Royer, and G. Salaün. SHE'S Project: a Report of Joint Works on the Integration of Formal Specification Techniques. In *Proc. of the Workshop on Integration of Specification Techniques with Applications in Engineering (INT'02)*, pages 29–36, France, 2002.
4. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P.D. Mosses, D. Sannella, and A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science*, 286(2):153–196, 2002.
5. E. Astesiano, M. Cerioli, and G. Reggio. Plugging Data Constructs into Paradigm-Specific Languages: Towards an Application to UML. In T. Rus, editor, *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 273–292, USA, 2000. Springer-Verlag.
6. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors. *Algebraic Foundations of System Specification*. Springer-Verlag, 1999.
7. C. Attiogbé, A. Francheteau, J. Limousin, and G. Salaün. ISA, a Tool for Integrated Specifications Animation. `ISA/isa.html` in Salaün's webpage.
8. C. Attiogbé, P. Poizat, and G. Salaün. Integration of Formal Datatypes within State Diagrams. Technical Report 83-2002, University of Evry, October 2002.
9. C. Attiogbé, P. Poizat, and G. Salaün. Specification of a Gas Station using a Formalism Integrating Formal Datatypes within State Diagrams. In *Proc. of the 8th International Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA'03)*, IEEE Computer Society Press, France, 2003. To appear.
10. R. Büssow and M. Weber. A Steam-Boiler Control Specification with Statecharts and Z. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler*, volume 1165 of *LNCS*, pages 109–128. Springer-Verlag, 1996.
11. M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1):55–61, 2002.
12. C. Choppy, P. Poizat, and J.-C. Royer. A Global Semantics for Views. In T. Rus, editor, *Proc of the 8th International Conference on Algebraic Methodology And Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 165–180, USA, 2000. Springer-Verlag.
13. J. Crow, S. Owre, J. Rushby, N. Shankar, and M. Srivas. A Tutorial Introduction to PVS. In *Proc. of the Workshop on Industrial-Strength Formal Specification Techniques (WIFT'95)*, USA, 1995. Computer Science Laboratory, SRI International.
14. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL : Formal Object-oriented Language for Communicating Systems*. Prentice-Hall, 1997.

15. S. J. Garland and J. V. Guttag. A Guide to LP, the Larch Prover. Technical Report, Palo Alto, California, 1991.
16. M. Große-Rhode. Integrating Semantics for Object-Oriented System Models. In F. Orejas, P.G. Spirakis, and J. van Leeuwen, editors, *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*, pages 40–60. Springer-Verlag, 2001.
17. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.
18. M. Hennessy and H. Lin. Symbolic Bisimulations. *Theoretical Computer Science*, 138(2):353–389, 1995.
19. J. Jürjens. A UML Statecharts Semantics with Message-Passing. In *Proc. of the 17th ACM Symposium on Applied Computing (SAC'02)*, pages 1009–1013, Spain, 2002. ACM Inc.
20. D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In P. Ciancarini and R. Gorrieri, editors, *Proc. of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, pages 331–347, Italy, 1999. Kluwer Academic Publishers.
21. J. Lilius and I.P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *Proc. of the International Conference on the Unified Modelling Language: Beyond the Standard (UML'99)*, volume 1723 of *LNCS*, pages 430–445, USA, 1999. Springer-Verlag.
22. G. Reggio and L. Repetto. Casl-Chart: A Combination of Statecharts and of the Algebraic Specification Language Casl. In T. Rus, editor, *Proc. of the 8th International Conference on Algebraic Methodology and Software Technology (AMAST'00)*, volume 1816 of *LNCS*, pages 243–257, USA, 2000. Springer-Verlag.
23. E. Sekerinski and R. Zurob. Translating Statecharts to B. In M. Butler, L. Petre, and K. Sere, editors, *Proc. of the 3rd International Conference on Integrated Formal Methods (IFM'02)*, volume 2335 of *LNCS*, pages 128–144, Finland, 2002. Springer-Verlag.
24. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.
25. M. van der Beeck. Formalization of UML-Statecharts. In M. Gogolla and C. Kobryn, editors, *Proc. of the 4th International Conference on the Unified Modelling Language (UML'01)*, volume 2185 of *LNCS*, pages 406–421, Canada, 2001. Springer-Verlag.

Xere: Towards a Natural Interoperability between XML and ER Diagrams*

Giuseppe Della Penna, Antinisca Di Marco, Benedetto Intrigila, Igor Melatti, and Alfonso Pierantonio

Università degli Studi di L'Aquila, Dipartimento di Informatica
{dellapenna,adimarco,intrigila,melatti,alfonso}@di.univaq.it

Abstract. XML (eXtensible Markup Language) is becoming the standard format for documents on Internet and is widely used to exchange data. Often, the relevant information contained in XML documents needs to be also stored in legacy databases (DB) in order to integrate the new data with the pre–existing ones. In this paper, we introduce a technique for the automatic XML–DB integration, which we call Xere. In particular we present, as the first step of Xere, the mapping algorithm which allows the translation of XML Schemas into Entity-Relationship diagrams.

1 Introduction

Over the last years, Internet related technologies had an exponential growth and motivated a stronger demand for standards in information interchange and treatment. As a consequence, using XML (eXtensible Markup Language) [8] as a standard format for data and documents on Internet is becoming a commonplace.

Moreover, we are facing the shift of XML usage from its original *presentation-centric* nature to a more *information-centric* one. A number of organizations and individuals are using XML to exchange data that need to be stored and managed in some way. Information is often originated directly in XML, and since XML data are contained in files, it is usual to store them "as they are" on a filesystem. Nevertheless, the relevant information contained in such files often needs to be also stored in (legacy) databases. Indeed, applications that manage and manipulate this information usually work on (relational) databases, and companies want to continue to use these "legacy" applications, since adapting them to read directly data from XML files may be too expensive and complex. This results in an almost unavoidable duplication of data, which is often troublesome: the association between the XML data and its image in the database is possibly not preserved, and this may lead to consistency errors.

To face these problems, we propose a general technique, which we call Xere (XML Entity Relationship Exchange), to assist the XML – DB integration, in a way that alleviates the consistency problems and helps in merging the XML data with pre–existing legacy ones. In particular, the database structures that contain the XML data should be naturally related to the XML document structure, so

* Research partially supported by the MURST project SAHARA

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 356–371, 2003.

that it could be possible to transparently store XML data in databases, query the resulting data structures, and automatically regenerate "on the fly" the source documents from the DB content.

In this paper, we present the first step of Xere, consisting of an algorithm that maps XML Schemas to Entity–Relationship models.

Mapping XML Schemas to the Entity–Relationship conceptual model, rather than to the relational structures of a database, has various advantages:

- The ER diagram offers a good documentation to the DB designer and maintainer.
- A *natural* mapping to the relational model is not always possible, since the relational model and XML Schemas are on two very different abstraction layers. In particular, there are many constraints and optimizations that can be only "seen" on the ER model, due to the part of structural information which is lost in the "flat" relational model. Indeed, in the traditional database design process, the relational model is used only in one of the last steps, very close to the physical deployment.
- The integration with pre–existing data structures in a legacy database can be more easily studied on a conceptual model.

We decided to adopt XML Schemas [9] rather than DTDs for a number of reasons. Although, previous approaches (such as [11]) to the XML–DB integration use DTDs to create a general mapping algorithm that is subsequently applied to the XML documents, DTDs are being progressively replaced by XML Schemas.

Moreover, using XML Schemas is convenient since this formalism offers a better expressiveness to describe very complex document structures. It can express advanced concepts like generalization, type derivation and substitution, complex type definitions and a variety of content models (such as sequence, choice, set). XML Schemas are also strongly typed – as databases are: actually, the XML Schema basic types are directly derived from SQL data types (see [9]), and this helps also in the physical deployment of the database structures.

Finally, since a XML Schema is itself an XML-based formalism, it can be processed by a variety of XML–based tools. Indeed, in this paper we also present the prototypical implementation of our Schema to ER mapping using only XML-related technologies and (free) tools. Note that many free tools are also available to translate DTDs in XML Schemas [9], so our approach could indirectly handle also DTDs.

The paper is arranged as follows. Section 2 presents the Xere mapping outlining how XML Schemas are translated into Entity-Relationship diagrams. In Sect. 3 we prove the completeness and soundness of the Xere mapping algorithm. Finally, Sect. 5 compares the work which is presented in this paper with related works and Sect. 6 draws some conclusions and discusses future works.

2 Xere: Mapping XML Schemas into ER Diagrams

This section illustrates the Xere mapping. Since XML Schemas are also XML documents, a running implementation of Xere is given by means of XSLT [10],

as described in Sect. 2.2. In Sect. 2.3 we also sketch a possible optimization of the mapping.

2.1 Mapping Algorithm

The mapping algorithm takes an XML Schema as input and creates the corresponding Entity-Relationship (ER in the following) model as output.

The mapping algorithm illustrated in this section does not support all the features of XML Schemas, because of space limitations, and we focus on a meaningful subset of Schema elements.

Table 1 shows all the XML Schema elements, divided in categories, and some of their attributes. For a complete reference on the XML Schema grammar and its semantics, the reader can refer to [9]. The "Supported" columns in Table 1 show which elements are supported in the current implementation of the Xere mapping. The omitted elements are not fundamental for the schema definitions. Therefore, our current algorithm can be considered effective on most of the XML Schemas. In the following, we describe the features recognized by the mapping algorithm and the resulting ER structures.

Table 1. Schema elements and attributes currently supported by the Xere mapping

Category	Element	Supported
Particles	all	Yes
	element	Yes
	choice	Yes
	sequence	Yes
	group	Yes
	any	No
Attributes	anyAttribute	No
	attribute	Yes
	attributeGroup	Yes
Complex types	complexType	Yes
	complexContent	Yes
	simpleContent	Yes
	restriction	Yes
	extension	Yes
Simple types	simpleType	No
	list	
	union	
Identity constraints	unique	No
	key	Yes
	keyref	Yes
	field	Yes
	selector	Yes
Other elements	schema	Yes
	notation	No
Facets	maxExclusive	No
	maxInclusive	
	minExclusive	
	minInclusive	
	length	
	maxLength	
	minLength	
	fractionDigits	
	totalDigits	
	pattern	
	enumeration	
	whiteSpace	
Comments	annotation	No
	appinfo	
	documentation	
Schema combination	import	No
	include	
	redefine	
Schema attributes	minOccurs	Yes
	maxOccurs	Yes
	mixed	No
	substitutionGroup	No
	default	No
	fixed	No
	nillable	No

```
<element name="A">
 <complexType>
  <sequence>
   <element name="B" type="integer"
      minOccurs="0"
      maxOccurs="unbounded"/>
  </sequence>
 </complexType>
</element>
```

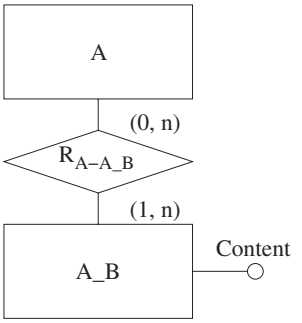


Fig. 1. Mapping of a local element.

```
<element name="A">
 <complexType>
  <sequence>
   <element ref="B"
      minOccurs="b1"
      maxOccurs="b2"/>
   <element ref="C"
      minOccurs="b3"
      maxOccurs="b4"/>
  </sequence>
 </complexType>
</element>
```

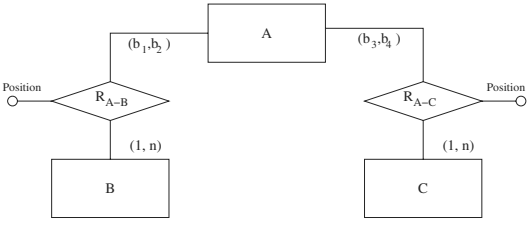


Fig. 2. Mapping of a sequence model.

```
<element name="A">
 <complexType>
  <choice minOccurs="b1"
          maxOccurs="b2">
   <element ref="B"
      minOccurs="b3"
      maxOccurs="b4"/>
   <element ref="C"
      minOccurs="b5"
      maxOccurs="b6"/>
  </choice>
 </complexType>
</element>
```

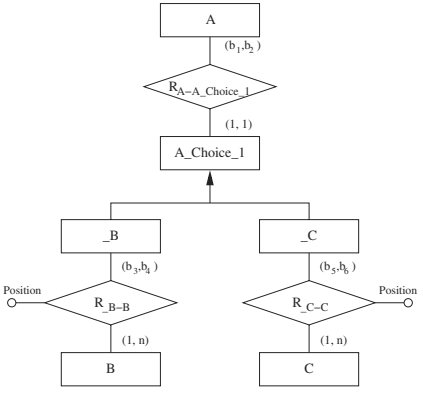


Fig. 3. Mapping of a choice model.

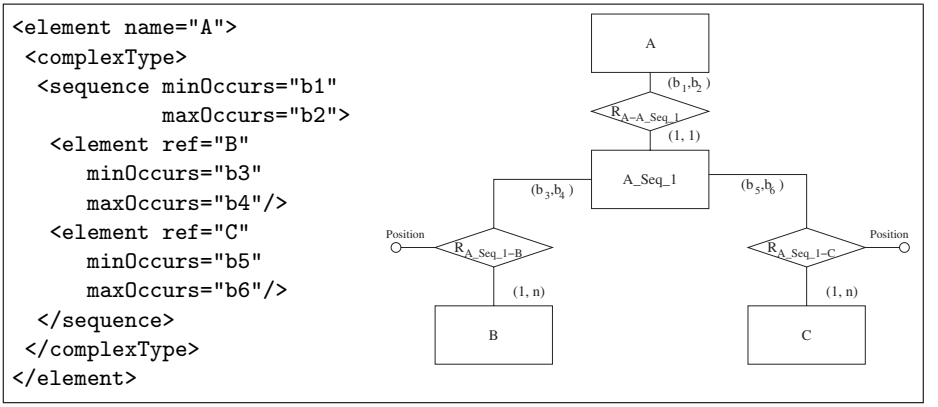


Fig. 4. Mapping of a sequence model with occurrence constraints using an auxiliary entity.

Elements. Each element becomes an entity. The entity name is obtained from the element name considering its nesting (see "Element nesting" below). The element primary key is chosen using the following rules:

1. if the schema explicitly defines a key (using the `xs:key` construct[1]) for the element, that key is used as its primary key. Note that the keys defined using the `xs:key` construct can be composite and make references to attributes belonging to other elements (which are parents of the current one). In this case, we are simply declaring the resulting entity as a weak entity, so its primary key depends on some attributes of other entities that are associated with it via a parent relationship. Our algorithm guarantees that the relation between any element and its parent exists in the resulting ER model.
2. if the element has an attribute of type `xs:ID`, that attribute becomes its primary key.
3. otherwise, the entity has not any key. Entities with no keys are secondary entities that will never be addressed directly.

Simple elements (e.g. without attributes and child elements) have a special "content" attribute, which holds the textual content of the element. Simple elements may be further optimized (see Sect. 2.3).

Attributes. Element attributes are mapped to entity attributes, regardless of their `xs:use` attribute value (e.g. required, implied, etc.).

Element Nesting. Elements in XML Schemas can have a scope, so e.g. we can define elements with the same name in different types. The mapping takes care of this situation by using local names when creating entities derived from nested definitions. The local name is simply obtained by appending the element name to the container's name: for example, an element "B" declared inside another element "A" will result in the creation of an entity named "A_B" (see Fig. 1 for an example).

[1] In this paper, the `xs` prefix always denotes the XML Schema namespace.

Global Declarations and References. Elements, attributes and types can be globally declared and then referenced by name. The algorithm handles both situations.

- global types are inline expanded where referenced.
- global elements are created once (as entities) and referenced when needed.
- global attributes are copied in any entity that references them.

Complex Types. All kind of complex types are recognized. Types derived by extension from other complex or simple types are expanded and the final type is obtained merging all the types in the derivation hierarchy. Types derived by restriction are handled as natural by rewriting the entire complex type with appropriate restrictions.

Content Models. All content models (i.e. `sequence`, `choice`, `all`) are recognized. The mapping is designed to be as natural as possible. Content children can be either elements or other content models. In general, all the content children are created applying recursively the mapping algorithm and then attached to the content owner using relations. Note that, in our ER models, relations are oriented: this means that we can always say that a particular relation *exits* from an entity (the "parent") and *is directed to* another entity (the "child"). Moreover, since these relations represent a parent–child relationship in the XML Schema, their cardinalities are $(1 \text{‹} 1)$ on the parent side, unless different occurrence constraints are explicitly defined, and $(1 \text{‹} n)$ on the child side, since children with the same content from different XML instance documents may be merged and attached to many parents for data optimization purposes. The particular mapping for each content model is described in the following paragraphs.

`Sequence` **Models.** All the content children are linked to the model owner using relations. The sequence relative positions of child elements are expressed in the ER diagram by adding a `position` attribute to these relations (see Fig. 2 for an example). Note that this "extra" attribute actually expresses an implicit attribute of the XML Schema: indeed, the choice of a sequence content model implies an ordering on the information represented by the model children.

`Choice` **Models.** The choice model is translated using the generalization construct of ER diagrams. An auxiliary entity is attached to the model owner with an appropriate relation, and is specialized in as many other auxiliary entities as the content children are. The reason for adding the auxiliary entities is that we need the corresponding relations to store the cardinality constraints. Finally, the actual content model children are linked to the corresponding auxiliary entity using a relation. If a child has occurrence constraints, we add to the corresponding relation a `position` attribute to keep track of the occurrences ordering (see Fig. 3 for an example). Note that the ER generalization construct may not be always semantically consistent with the choice XML Schema content model. However, the generalization, used in the particular way explained

above, is the only construct that allows to maintain at least the "mutual exclusion" semantics of the choice model in the ER diagram.

`All` **Models.** The all model is realized similarly to the sequence model, without keeping track of the children position in the sequence.

Elements `xs:group` are used to import frequently used content models. The algorithm expands them inline anywhere they are referenced. The expanded content models are recursively processed using the previously described rules.

Occurrence Constraints. Occurrence constraints in content model children are mapped to relation cardinalities in the ER model. As already said, a special attribute `position` is added to the relations to keep track of the occurrences ordinal position in choice and sequence content models. If the occurrence constraint is placed on a sequence model, an auxiliary entity and a relation are created to store the corresponding cardinality (see Fig. 4). Note that the role of auxiliary entities is only to provide a more natural way to map the Schema to the ER diagram. Most of the times, these temporary entities are discarded when reducing the ER schema to a relational schema.

Key De nitions and Key References. Key definitions are mapped as entity primary keys. Key references are translated by executing the two following steps:

1. the attribute corresponding to the `xs:field` part of the key reference is removed from its entity.
2. a relation, with the same name of the removed attribute, is created from the referring entity to all the entities containing the attributes that compose the key. The cardinalities of this relation are $(1 \text{‹} 1)$ on the side of the entity with the removed attribute and $(1 \text{‹} n)$ on the other sides.

Finally, the ER model is completed by creating an additional entity, conventionally called *document*, that has a single attribute called *name*. This entity is associated through a relation to the entity representing the XML document root. In this way, we can distinguish the elements of each different document stored in our data base.

2.2 An XSLT Implementation of the Mapping

In this section we briefly introduce a prototype implementation of most of the Xere mapping algorithm described in Sect. 2, with the exception of primary keys, key references, attribute and element groups.

We implemented the algorithm using a XSLT [10] transformation stylesheet that can be used and tested on possibly any platform using an XSLT interpreter.

The output of the transformation is another XML document describing the resulting ER model with a very simple markup containing the following elements:

- `<entity name="foo">` declares an entity of the ER diagram called `foo`.
- `<attribute name="foobar" type="atype">` inside an `<entity>` or `<relation>` element, declares an attribute of that entity or relation with the specified name and type.

- <relation name="foobar" from="foo" to="bar" card="n_1,m_1-n_2,m_2"> declares a relation, called foobar, between the two named entities foo and bar, with the given cardinality (n_1,m_1 and n_2,m_2 are the cardinalities of the foo and bar entities in the relation, respectively).
- <specialization base="foo"> defines a generalization having foo as its base (most general) entity. All the children of the <specialization> element are possible specializations of the base.

The complete implementation of the mapping algorithm outlined here, together with the XML Schema of the language used to represent the ER models, can be found in [3]. Moreover, it is possible to try the algorithm online at the url http://dellapenna.univaq.it/xere/.

2.3 Optimizations on the Algorithm

Many optimizations can be applied to the mapping algorithm described in Sect. 2.1. We decided to remove these optimizations from the first release of the algorithm, since they may make the resulting ER diagram more complex and difficult to understand.

We may suggest two major optimization to the current Xere mapping algorithm:

- Optimizations based on cardinalities and simple types may decrease the number of entities created by the algorithm. For example, an element with a simple type may be safely translated in an attribute (possibly with cardinality) if it is contained in another entity and has "reasonably low" occurrence constraint. However, this optimization should be carefully validated by domain experts not to alter the diagram semantics.
- Element groups are expanded inline in the current mapping. Therefore, the group content becomes a nested content model, and this may lead to the creation of many different instances of the same set of entities. To optimize this process, we may create a temporary entity called, say, *GroupX* and connect the group content model to it. Then, any reference to that group could be translated into a simple relation to the *GroupX* entity, thus saving much space. This transformation is semantically consistent since elements are grouped when they have a fixed semantics that is shared among several different content models.

3 Soundness and Completeness of the Xere Mapping

The Xere mapping algorithm described in Sect. 2 covers the main XML Schema elements, as shown in Table 1. There are some secondary elements and constructs we decided to not include in the first version of the Xere algorithm, since they are not fundamental for the schema definitions. Therefore, our current algorithm can be considered effective on most of the current XML schemas.

In this section we sketch the completeness and soundness proofs for the current Xere algorithm.

Proposition 1 (Xere Mapping Completeness). *The Xere mapping algorithm is complete w.r.t. the given subset of the W3C XML Schema Specification [9].*

Proof. By analyzing the meta–schema given in [9], we can see that the meta–schema is built recursively from the base elements described in Table 1. The Xere algorithm is well-defined on all the base elements shown in Table 1. These base elements are translated in ER entities. The algorithm is in turn recursive, so it can go down through the schema definition, translating the parent–child relations in ER relations. □

To prove the soundness of the Xere mapping we show that, given any XML document valid w.r.t. a particular schema, we can store the document content as an instance of the ER model generated by the Xere algorithm and then rebuild that document (or an equivalent instance) from that ER model instance.

In other words, we can represent XML documents as instances of the ER model created by the Xere mapping for the corresponding schema, so that the *document identity* is preserved.

De nition 1 (Document Identity). *We say that two XML documents D_1 and D_2 with the same schema S are* identical *(or that they have the same* identity*) iff they have*

- *The root elements of D_1 and D_2, namely R_1 and R_2, are the same (i.e. they have the same tag).*
- *The two root elements R_1 and R_2 have the same attributes with the same values. Attribute order is not considered. Attribute values should be expanded w.r.t. their defaults, as described in S.*
- *If R_1 and R_2 have a simple content, then the contents are identical.*
- *If R_1 and R_2 have a complex content, then there exists a bijective mapping between the children of R_1 and R_2, so that the document fragments corresponding to related children are in turn identical. In addition, if R_1 and R_2 have a sequence model (declared in the schema), then the bijective mapping must also preserve the child positions: the first child of R_1 is mapped into the first child of R_2, and so on.*

Proposition 2 (Xere Mapping Soundness). *The Xere mapping algorithm preserves the XML documents identity.*

Proof. Since the Xere algorithm is recursive, the soundness proof will be given by induction on the XML documents structure.

The simplest XML document is composed by a single element, possibly with attributes and a textual content. In an XML Schema, such element would be declared with a schema fragment like that shown in Fig. 5a.

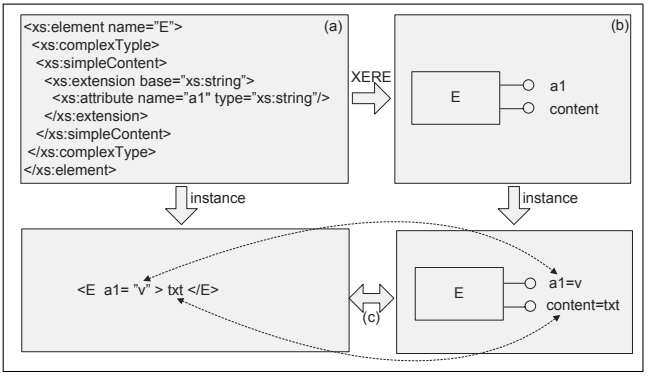


Fig. 5. Mapping of a simple element.

The Xere algorithm would translate this schema fragment creating an entity called E with two attributes, *content* and $a1$, which correspond to the element textual content and to the element attribute $a1$, respectively (see Fig. 5b).

An instance of the given schema is `<E a1=□□v□□>txt</E>`. This instance is mapped on the ER model as an instance of the entity E, where the entity attributes *content* and $a1$ are assigned to the values `txt` and `v`, respectively (see Fig. 5c).

From the other end, given an instance of the ER model described above, we map it to an XML document instance by creating an element with the same name of the entity, i.e. E, and assign it the textual content given by the *content* attribute, if it exists. Then we add an attribute to the element for each other attribute of E, and assign it with the corresponding value (see Fig. 5c).

For the ER model instance generated above, we would rebuild exactly the source XML `<E a1=□□v□□>txt</E>`.

Now we describe how content models are mapped. We inductively suppose that we can store and retrieve any XML subtree, and show how the content models (all, sequence, choice) are mapped.

A basic sequence model is declared by the schema fragment shown in Fig. 6a, where `M1`, `M2`, ..., `Mn` can be element declarations (or references) or nested content models.

The Xere algorithm would translate this schema fragment creating an entity called E, recursively building the ER sub–diagrams corresponding to `M1`, `M2`, ..., `Mn`, and connecting these sub–diagrams to E with n relations. The created relations have a *position* attribute that specifies the sequence position of the corresponding sub–diagram (see Fig. 6b). Note that, by induction hypothesis, we can always build the required sub–diagrams.

An instance of the given schema is `<E><E1>...</E1>...<En>...</En></E>`, where the subtrees `E1`, ..., `En` are valid w.r.t. the respective schema fragments `M1`, ..., `Mn`. This instance is mapped on the ER model instance by

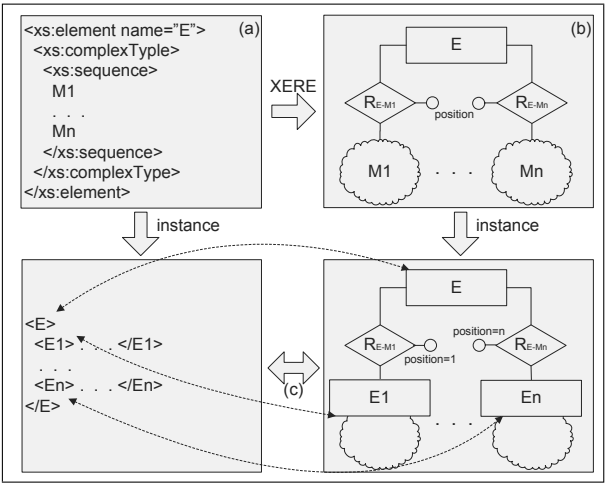


Fig. 6. Mapping of a sequence model.

1. first recursively creating the instances of the ER sub–diagrams corresponding to `E1`, ..., `En`;
2. then, an instance of the ER entity E is created and the relations are set to associate this instance with the sub–diagrams of $E1$, ..., En. Each relation instance has an attribute *position* that is set to the ordinal position of the corresponding sequence child `E1`, ..., `En` (see Fig. 6c).

On the other hand, given an instance of the ER model described above, we map it to an XML document instance by creating an element with the same name of the root entity, i.e. E, and recursively rebuilding the XML subtree corresponding to the n sub–diagrams associated to E with the n relations. Then, we insert these subtrees in the root element E using the order given by the *position* attribute on the relations (see Fig. 6c).

For the model instance generated in the example above, we would rebuild exactly the source XML `<E><E1>...</E1>...<En>...</En></E>`.

A basic choice model is declared by the schema fragment shown in Fig. 7a, where `M1`, `M2`, ..., `Mn` can be element declarations (or references) or nested content models.

The Xere algorithm would translate this schema fragment creating an entity called E and recursively building the ER sub–diagrams corresponding to to `M1`, `M2`, ..., `Mn`. A temporary entity T is created and associated with E using a relation R_{E-T}, and n other temporary entities T_M1, ..., T_Mn are attached as specializations of T. Finally, the roots of the sub–diagrams for `M1`, `M2`, ..., `Mn` are attached to the corresponding temporary entities T_M1, ..., T_Mn using an appropriate relation (see Fig. 7b).

An instance of the given schema is `<E><Ei>...</Ei></E>`, where the subtree `Ei` is valid w.r.t. the respective schema fragment `Mi`. This instance is mapped on

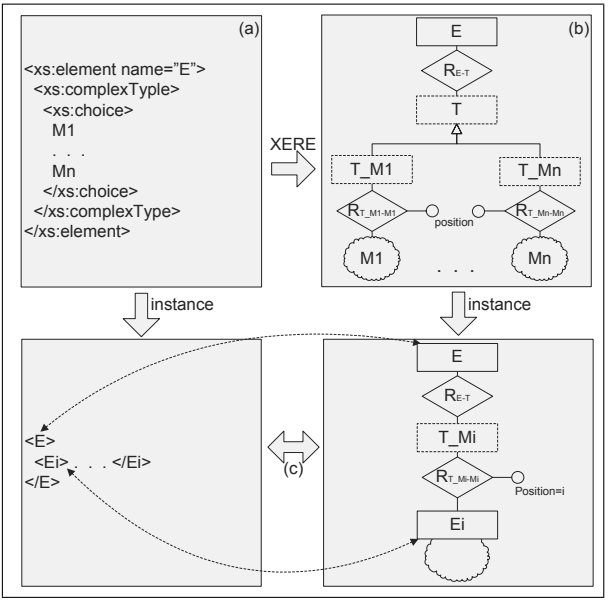


Fig. 7. Mapping of a choice model.

the ER model by first recursively creating the instance of the ER sub–diagram corresponding to `Ei`. Then, an instance of the ER entity E is created and the relation R is set to associate this instance with the sub–diagram of Ei, implicitly choosing the corresponding specialization of T (see Fig. 7c).

From the other end, given an instance of the ER model described above, we map it to an XML document instance by creating an element with the same name of the root entity, i.e. E, and recursively rebuilding the XML subtree corresponding to the specialized sub–diagram associated to E with the relation R. Then, we insert this subtree in the root element E (see Fig. 7c).

For the model instance generated in the example above, we would rebuild exactly the source XML `<E><Ei>...</Ei></E>`.

The all content model is mapped exactly as the sequence model, with the exception of the *position* attribute that is not created and ignored during the document storing and retrieval. Therefore, the all model mapping is even simpler than the sequence model mapping. □

Although we only give here a sketch of the inductive soundness proof, we think it should be enough to ensure the soundness of the Xere approach.

4 An Example

In this section we briefly show an application of our mapping algorithm and the obtained results. The Schema we used, shown in Figure 8, defines the structure

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           elementFormDefault="qualified">
 <!--Global Types: we skip the part where the global types "full_name_type",
     "name_type","authors_type","keywords_type","relatedwords_type" and
     where the global elements "family","middle" and "first" were defined-->
 <!--Root Element-->
 <xs:element name="issue">
  <xs:complexType><xs:sequence>
    <xs:element name="editor" type="name_type"/>
    <xs:element name="articles"><xs:complexType><xs:sequence>
      <xs:element ref="article" maxOccurs="unbounded"/>
     </xs:sequence></xs:complexType></xs:element>
  </xs:sequence></xs:complexType>
 </xs:element>
 <xs:element name="article"><xs:complexType><xs:all>
  <xs:element name="title" type="xs:string"/>
  <xs:element name="authors" type="authors_type"/>
  <xs:element name="summary">
   <xs:complexType><xs:choice>
     <xs:element ref="keywords"/>
     <xs:element ref="related_words"/>
   </xs:choice></xs:complexType></xs:element></xs:all>
  <xs:attribute name="category" type="xs:string"/>
  </xs:complexType></xs:element>
 <xs:element name="author" type="full_name_type"/>
 <xs:element name="keywords"><xs:complexType><xs:complexContent>
    <xs:extension base="keywords_type"/>
  </xs:complexContent></xs:complexType></xs:element>
 <!--We skip the definition of "related_words", that is similar to the
     "keywords" definition above-->
</xs:schema>
```

Fig. 8. XML Schema code for the journal issue example.

of a journal issue. We run on this Schema the stylesheet mentioned in Section 2.2 and the ER model resulting is shown in Figure 9.

Let us highlight the two most interesting parts of this Schema:

- the `articles` definition (exemplifying the `sequence` model);
- the `summary` definition (exemplifying the `choice` model)

that determined the generation of the shaded areas in Figure 9. Note that, since the `articles` element is nested in the `issue` element definition, the corresponding entity is labelled `issue_articles` in the generated ER diagram. For the same reason, the `summary` element nested inside the `article` element generates an entity labelled `article_summary`.

5 Related Works

Most of the related works are dealing with some kind of translation from XML to database systems. As stated in Sect. 2, current approaches make use of DTDs and directly map them on relational schemas. Therefore, we were not able to found other techniques directly comparable with ours. In this section we list general

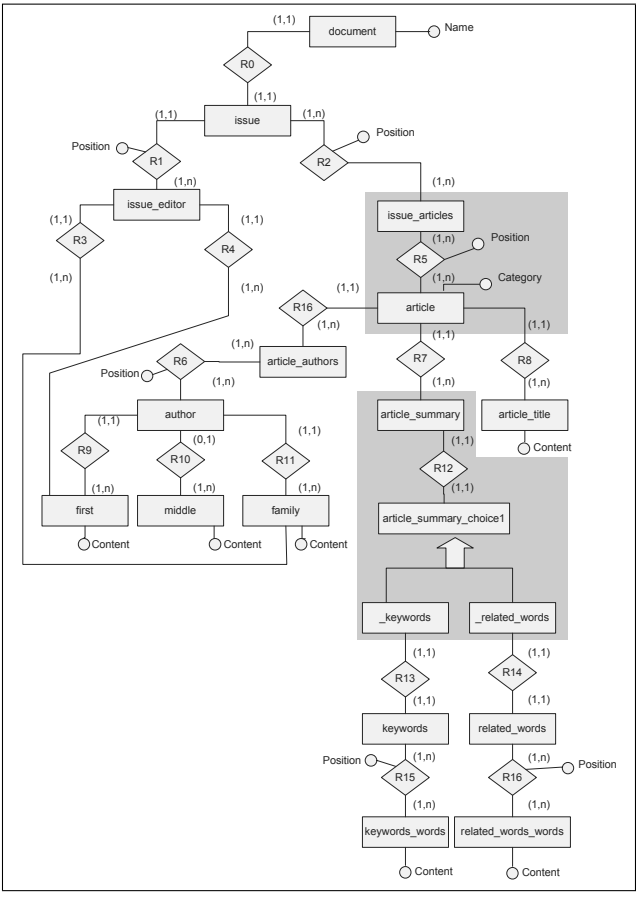


Fig. 9. The ER Model diagram for the journal issue example.

related work about XML–DBMS mapping. Note that many papers in this field still address SGML as the standard formalism for the definition of structured documents, so they actually talk about SGML–DBMS mappings. However, since XML is a subset of SGML, the SGML–DBMS mapping techniques explained in these papers also apply to XML.

There are two main approaches to designing relational database schemas for XML documents. The first approach, namely the *structure-mapping approach*, create relational schemas based on the structure of XML documents (deduced from their DTD, if available). Basically, with this approach a relation is created for each element type in the XML documents, [2,1], and a database schema is defined for each XML document structure or DTD. This is the approach we used in the Xere algorithm. Using ER as the target model, we can optimize the transformation and merge the obtained schemas with legacy DBMS structures.

More sophisticated mapping methods have also been proposed, where database schemas are designed based on detailed analysis of DTDs, [7].

In the *model-mapping approach,* a fixed database schema is used to store the structure of all XML documents. Basically, this kind of relational schema stores element contents and structural rules of XML documents as separate relations. Early proposals of this approach include, [12], or the "edge approach", [4], in which edges in XML document trees are stored as relational tuples. A more recent research using this approach is [11], that also defines an efficient method to query this kind of structure.

In both the approaches above, XML documents are decomposed into fragments and distributed in different relations. Obviously, these decomposition approaches have drawbacks – it takes time to restore the entire or a large subportion of the original XML documents. A simple alternative approach, supported in almost all the XML-enabled RDBMS (e.g. Oracle, SQL Server, etc.), is to store the entire text of XML documents in a single database attribute as a CLOB (Character Large Object). On the other hand, this approach does not allow queries on the document structure using SQL (since all the document is stored in a single field), and the search for a particular document node always implies loading all the XML text and searching using regular expression- or XPath-based engines.

Moreover, since SGML (Standard Generalized Markup Language), [5], was a predecessor of XML, there were several studies on the management of structured documents even before XML emerged, [6]. These methods can roughly be classified into two categories: a *database schema designed for documents with DTD* information and a *storage of documents without any information about DTDs.* The latter approaches are capable of storing well-formed XML documents that do not have DTDs. For both approaches, queries on XML documents are converted into database queries before processing. First, there are simple methods that basically design relational schemas corresponding to every element declaration in a DTD, [2,1]. Other approaches design relational schemas by analyzing DTDs more precisely. An approach to analyze DTD and automatically convert it into relational schemas is proposed in [7]. In this approach, a DTD is simplified by discarding the information on the order of occurrence among elements.

6 Conclusions and Future Work

In this paper, we presented the first step of the Xere methodology, namely a rule-based process to translate an arbitrary XML Schema to an Entity–Relationship diagram in a natural way. This mapping is shown to be sound and complete w.r.t. document identity and XML Schema definition, respectively.

The mapping is denoted as *natural* since it preserves all hierarchical relations defined by a XML Schema. Essentially, the mapping is able to retain all the structure defined by a XML Schema encoding it in the relations and entities of the generated ER diagram. We intend to further investigate this compatibility property as, we believe, it will lead to a homomorphism definition.

In this work, we introduced only the first step of the Xere technique. Next efforts will be devoted to achieve the complete interoperability between XML and relational databases. This requires a translation of ER schemas into relational models and a procedure which allows one to transparently store and retrieve XML documents in the databases created using the Xere technique.

References

1. Abiteboul, S., Cluet, S., Christophides, V., Milo, T., Moerkotte, G., and Siméon, J. 1997. Querying documents in object databases. *Int. J. Dig. Lib.* 1, 1, 5–19.
2. Christophides, V., Abiteboul, S., Cluet, S., and Scholl, M. 1994. From structured documents to novel query facilities. *SIGMOD Rec. 2*3, 2 (June), 313–324.
3. Della Penna, G., Di Marco, A., Intrigila B., Melatti, I., Pierantonio, A. 2002. Towards the expected interoperability between XML and ER diagrams. Technical Report TRCS/G0102, Department of Computer Science, University of L'Aquila.
4. Florescu,D. and Kossmann, D. 1999. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Tech. Bull. 2*2, 3, 27–34.
5. ISO. 1986. Information processing—Text and office systems—Standard General Markup Language (SGML). ISO-8879.
6. Navarro,G. and Baeza-Yates, R. 1997. Proximal nodes: A model to query document databases by content and structure. *ACM Trans. Inf. Syst. 1*5, 4, 400–435.
7. Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., Dewitt,D.J., and Naughton, J.F. 1999. Relational databases for querying XML documents: Limitations and opportunities. In *Proceedings of the 25th International Conference on Very Large Data Bases* (VLDB, Edinburgh, Scotland, Sept. 7-10). Morgan Kaufmann, San Mateo, CA, 302–314.
8. World Wide Web Consortium. 1998. eXtensible Markup Language (XML) 1.0. `http://www.w3.org/TR/1998/REC-xml-19980210`
9. World Wide Web Consortium. 2001. XML Schema. `http://www.w3.org/XML/Schema`
10. World Wide Web Consortium. 2001. The eXtensible Stylesheet Language (XSL). `http://www.w3.org/Style/XSL/`
11. Yoshikawa, M. and Amagasa, T. 2001. XRel: A Path-Based Approach to Storage and Retrieval of XML Documents Using Relational Databases. *ACM Transactions on Internet Technology, 1,* 110-141
12. Zhang, J. 1995. Application of OODB and SGML techniques in text database: an electronic dictionary system. *SIGMOD Rec. 2*4, 1 (Mar.), 3–8.

Detecting Implied Scenarios Analyzing Non-local Branching Choices

Henry Muccini

Universitá degli Studi dell'Aquila
Dipartimento di Informatica
Via Vetoio, 1 – L'Aquila*
muccini@di.univaq.it

Abstract. Scenarios are powerful tools to model and analyze software systems. However, since they do not provide a complete description of the system, but just some possible execution paths, they are usually integrated with state machines. State machines may be extracted from scenarios using a synthesis process. We could expect that the synthesized state machine model is "equivalent" to the original scenario specification. Instead, it has been proven that it does not always hold, and state machines may introduce unexpected behaviors, called *implied scenarios*. This paper proves that there is a strict correlation between implied scenarios and non-local branching choices. Based on this result, we propose an approach to identify implied scenarios in High-Level Message Sequence Chart specifications and its application to some specifications. We finally highlight advantages with respect to existent approaches.

1 Introduction

Scenarios describe a temporal ordered sequence of events and are used by researchers and industry practitioners for a variety of purposes and at different phases in the software development process (e.g., [11]).

Although very expressive, this approach has two drawbacks with respect to analysis and validation: *model incompleteness* [19] and *view consistency* [14,16]. In order to mitigate both problems, state machines, synthesized from scenarios, are used to complement scenario specifications. When a synthesis process is applied to a set of scenarios, one expects that the synthesized state machines correctly reflect the scenario specification. What may happen, instead, is that the state-based model synthesized from the system scenarios presents sets of behaviors that do not appear in the scenarios themselves. This result has been initially presented in [1,21] proving that the synthesized state machines may introduce unexpected behaviors called "implied scenarios".

Implied scenarios are not always to be considered wrong behaviors. Sometimes they may be just considered as unexpected situations due to specification incompleteness. Anyway, they represent an underspecification in the Message

* Currently on leave at the University of California, Irvine

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 372–386, 2003.

Sequence Chart (MSC) specification and they have to be detected and fixed before the synthesized state machine is used for further steps.

One of the contributions of this paper is the investigation of the relationship between scenarios and their synthesized state machine, by discovering that there is a close dependence between implied scenarios and non-local branching choice[1] [2]. Based on the result that implied scenarios depend on non-local choices, we propose an approach and an algorithm able to identify implied scenarios in specifications composed by both MSC and High-Level MSC (hMSC) notations. The main advantages are that we apply a *structural, syntactic*, analysis of the specifications as opposed to the *behavioral, model-based* analysis in [21], reducing time and space complexity. Moreover, since we do not need to create the synthesized model, we save computational time and prevent the possibility of state explosion problems. A current limitation of our approach is that it can be applied only to specifications composed by both MSCs and hMSCs while the approach presented in [1] can detect implied scenarios just starting from a set of MSCs. The approach is finally applied to some specifications and compared to others.

The paper is organized as follows. Section 2 presents an overview on MSC and hMSC. Section 3 describes what an implied scenario is and why it is introduced during the synthesis process. Section 4 proposes the approach, how it may be implemented and its application to the Boiler case study. Section 5 points out related work while Sect. 6 presents preliminary results. Section 7 concludes the paper by presenting future work.

2 Message Sequence Charts and High-Level Message Sequence Charts

Message Sequence Chart (MSC) is a graphical (and textual) specification language, initially introduced in the telecommunication industry to model the exchange of messages between distributed software or hardware components. In 1992 it has become ITU standard. The latest version is known as MSC-2000 [12].

A MSC (also called basic MSC, bMSC) describes, in a intuitive way, how a set of parallel processes[2] interact, displaying the order in which messages are exchanged. Simple MSCs are shown in Fig. 1: each rectangular box represents a process, the vertical axes represent the process life line, each arrow defines a communication between the left and the right processes (i.e., send and receive of a message), the arrow's label identify the exchanged message. Messages are communicated asynchronously. Many other features are included in the MSC-2000 specification and are documented in [12,17].

The MSC-2000 semantics can be sketched with the help of Fig. 1:
– the send of a message must occur before its reception. For example, message m1 (in MSC1, Fig. 1a) may be received from process P2, only if m1 has been previously sent by process P1;

[1] as we will see later, a non-local choice is a consequence of an under specification in MSC specifications and can give rise to unintentional behavior.

[2] or "instances", following the terminology used in [12].

– within each process, events are totally ordered according to their position in the component life line. Two messages, m1 and m2, are causally dependent if there is a process P so that m1 is sent to or received from P and m2 is sent to or received from P. For example, the send of message m4 in MSC2 (Fig. 1b) is causally dependent on the reception of message m3 (since P2 is related to both messages); message m4 in MSC3 (Fig. 1c) can be sent and received independently from messages m5 and m6;
– in between the send and the receive of the same message, other messages may be sent/received. For example, m2 in MSC1, Fig. 1a, may be sent before the reception of message m1, since they are non causally dependent.

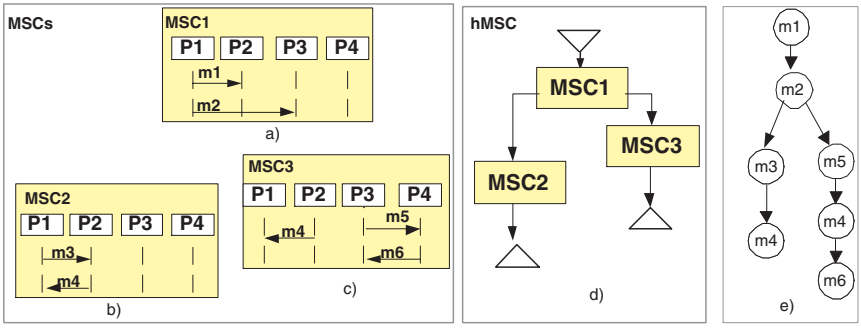


Fig. 1. a) MSCs, d) the hMSC diagram, c) the hMSC graph

To express more complex behaviors and support modularization of MSCs, **High-Level MSC (hMSC)** [12] may be used: they provide operators to link MSCs together in order to describe parallel, sequential, iterating, and non-deterministic executions of basic scenarios. In addition, hMSCs can describe a system in a hierarchical fashion by combining hMSCs within an hMSC.

In Fig. 1d the hMSC notation is shown. The upside down triangle indicates the start point, the other triangles indicate the end points and the rectangles identify MSCs. Figure 1d combines together MSCs in Figs. 1a, 1b, and 1c.

The ITU semantics for unsynchronised hMSCs identifies how MSCs can be combined. Given two MSCs, "MSCa" and "MSCb", and a hMSC in which MSCa precedes MSCb (MSCa -> MSCb):

1. MSCa -> MSCb does not mean that all events[3] in MSCa must precede events in MSCb;
2. an event "b" in MSCb can be performed before another event "a" in MSCa as soon as the processes involved in "b" are not anymore reacting in MSCa;
3. the second event in a generic MSC may be performed before the first one if the processes involved in the first communication are not involved in the

[3] the terms "event" and "action" will be used to identify messages exchanged in the MSCs.

second one. For example, m4 in MSC3 (Fig. 1c) may be sent before m5 in the same MSC.

A hMSC$_G$ graph will be used in the following to represent how actions are related together in the hMSC diagram (Fig. 1e): a node in the graph represents one communication inside the MSCs and it is labeled with the exchanged message name. An edge between node n1 and n2 is drawn when message n1 is exchanged before n2 in the same MSC or n1 is the last node in MSC1, n2 is the first node in MSC2 and MSC2 is reached by MSC1 in the hMSC. Figure 1e draws the hMSC$_G$ for the specification in Fig. 1d. In order to simplify this graph, we assume to deal with synchronous communication. Anyway, the graph can be used to represent bounded asynchronous communication (as in [1]) adding components to act as buffers.

A formal description of sequence chart semantics is out the scope of this paper and interested readers may refer to [12]. With the term *MSC specification* we will refer in the following to the hMSC specification together with its MSCs.

3 Implied Scenarios: The "What" and the "Why"

To explain *why* implied scenarios are introduced, we reuse the example in Fig. 1. The LTS synthesized from the MSC specification is shown in Fig. 2: each dashed box corresponds to one of the processes, the initial state is represented by a solid circle while the final one by a double circle.

As we can see in Fig. 2, m1.m2.m5.m3 is a feasible behavior in the synthesized state machines: P1 sends m1 to P2, then it sends m2 to P3. After the reception of m2, P3 can send m5 to P4. P1 can finally send m3 to P2.

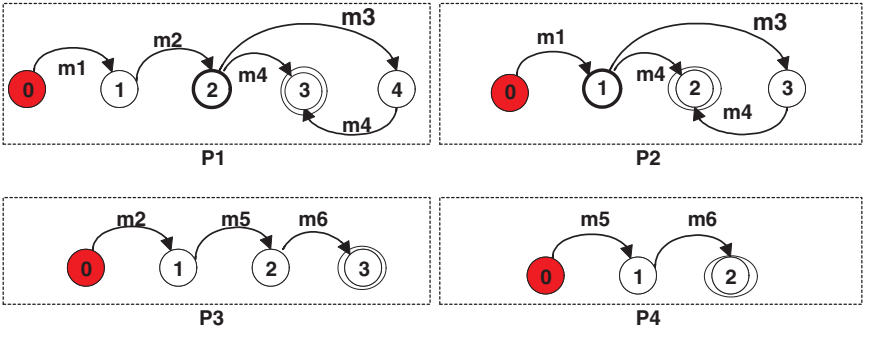


Fig. 2. The synthesized state machine for Fig. 1 specification

If we analyze how the MSC specification in Fig. 1 implements this behavior, we can notice that m1 and m2 are exchanged inside MSC1, then the system proceed to MSC3 with m5. To communicate via m3 (as expected by the

synthesized model), the flow should proceed to MSC2, but *this action is not allowed by the bMSC semantics* previously outlined. In fact, m4 in MSC3 should precede m3 in MSC2 since both of them rely on process P2 (hMSC semantics, second rule, Sect. 2). What we found is a flow of execution, allowed in the synthesized state machine but not specified in the MSC specification, that is, an implied scenario.

Now we will understand the *why* of implied scenarios: analyzing closely the synthesized state machines and the selected path, we can notice that after m2 is sent, (i.e., P1 is in state 2, P2 is in state 1 in Fig. 2) both P1 and P2 become ready to communicate via m3 and m4. After m5 has been exchanged between P3 and P4, P1 and P2 are *still* able to react using m3 while it is not permitted by the hMSC specification. If we take a look back to the MSC specification (Fig. 1) we can notice that this situation may arise when a branch is presented in the hMSC_G graph (node m2 in Fig. 1.e). In a branch, the system can evolve in different directions. Let b1, b2 and b3 be three possible branches. What may happen, in a branch, is that each process freely decides to take one of the possible branches and gets ready to react in that branch. When the system decides to evolve in one direction (e.g., branch b2), all the processes not involved in the first action in b2, do not react. Thus, they are not aware of the system's choice and they are still ready to react as nothing happened. As a result, when the system evolves in b2, those processes are still ready to act as in b1 or b3. This concept is usually known as "non-local branching choice" [2] and arises when there is a branch in the MSC specification, and the first events, in the different branches, are sent by different processes.

In our example, there is a branch in node m2 that generates a non-local choice. In fact, the first message in node m3 is sent by P1 while the first message in node m5 is sent by P3. If the system evolves in MSC3, process P3 sends the m5 message to P4. At this point, P1 and P2, that were not reacting, are still able to communicate via m3 and this communication creates the implied.

To have an implied scenarios these conditions hold: i) there is a non-local branching choice in the MSC specification so that ii) two processes maintain "extra" information[4] that can make them communicate in an unexpected way. We can summarize this section as follows:

– *What is an implied scenario?* An implied scenarios is a behavioral path that can be extracted from the state machine model but does not exist in the MSC specification.

– *Why do implied scenarios exist?* An implied scenario is due to a non-local choice situation in which two processes are enabled to communicate thanks to an extra information they catch. Notice that a non-local choice is not enough to have an implied scenario.

[4] this concept will be better explained later.

4 The Approach and the Algorithm

This section is utilized to describe the approach and the algorithm we propose to detect implied scenarios. Section 4.1 informally presents the overall idea. Section 4.2 refines the overall idea describing how the approach may be implemented by an algorithm which partially reuses existent tools. Section 4.3 shortly presents the *Boiler*[5] (v3) specification described in Fig. 3 and applies the approach to this example. Section 4.4 analyzes the algorithm completeness and correctness.

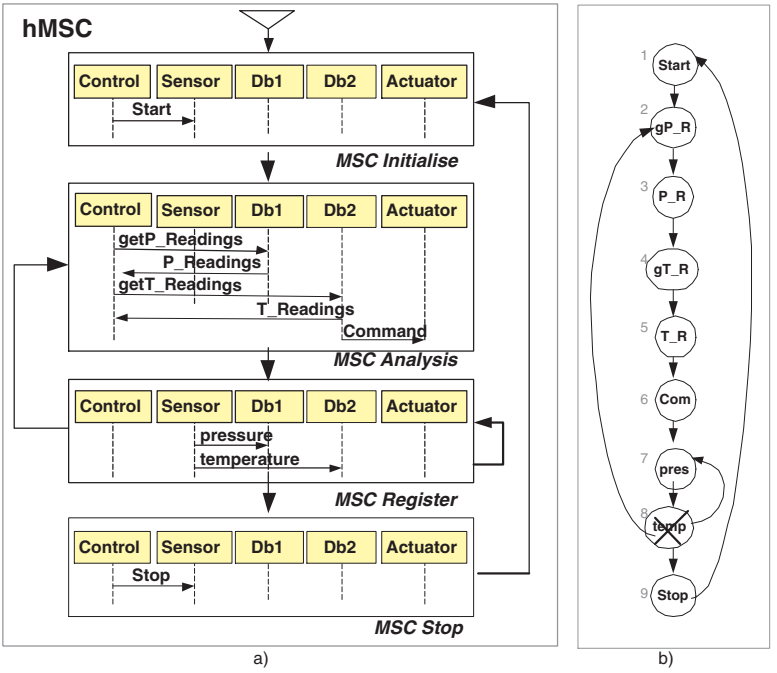


Fig. 3. a) The Boiler (version 3) Example, b) its hMSC_G graph

4.1 High Level Description

Briefly, an implied scenario may be found in the MSC specification when a non-local choice occurs that lets processes keep extra information that is lately used for a communication. Step 1 is used to detect the presence/absence of the non-local choice(s) in the hMSC_G graph. When a non-local choice arises, each process P gets ready to execute all possible branches and its state corresponds to the state P has in all the nodes directly reachable from the non-local choice. Step

[5] This specification comes from [22].

2 detects the state of each process after a non-local choice. Since P maintains this state in all the following hMSC$_G$ nodes where P does not react, we say that P1 keeps an "augmented" behavior that comes from the non-local choice. Step 3 identifies, for each process P and for each node n in the hMSC$_G$, the augmented of P in n. When two processes P1 and P2 share the same augmented behavior e in the same hMSC$_G$ node n, then their interaction generates the implied scenario. Step 4 detects the implied scenario.

4.2 The Algorithm

In this section we describe, using a more precise terminology, how the approach can be implemented through different steps.

Step 1: Non-local Choice Detection
The algorithm proposed in [2] and implemented inside the MESA framework [3] may be applied to detect non-local choices in MSC specifications. Informally, the tool examines the MSCs involved in a choice and verify that they all have the same unique process which sends the first event. If it does not apply, a non-local choice is identified and the hMSC nodes involved in the non-local branch are identified.

Technically, given the MSC specification, a direct graph (called Message Flow Graph (MFG)) is built and analyzed using the algorithm proposed by Ben-Abdallah and Leue [2]. The algorithm visits the graph an identifies the non-local choice nodes. Since the algorithm assumes that each process in each MSC exchange at least one message with other processes (assumption that does not apply in general), it needs to be slightly extended, as already pointed out in [2].

In the context of the hMSC$_G$ graph, we can define a non-local choice node as follows:

Def: *N-L Node*
A node $n0$ in the hMSC$_G$ is a non-local choice node (N-L node) if it is a branching node and the messages in DR(n) are sent by different processes.

where:

Def: *Direct Reachability and DR(n) Function*
We will say that nodes $n1$,..., nk are directly reachable from node n if they may be reached from n in through one edge on the hMSC$_G$.
The function DR: $n \longrightarrow$ N will return the nodes N directly reachable from n.

Step 2: State(P,N-L Node) Detection
If a non-local choice is detected, the algorithm needs to identify the state assumed by each process P in the N-L node. The following definitions will be useful for next discussions:

Def: *State of P in n and State(P,n) Function*
Let n be a node in the hMSC$_G$ graph and P a process. The state of P in n identifies the messages P can send or receive in n.
The function State: P x $n \longrightarrow s$ will return the state s of P in n.

Using the notation above, we need to compute State(P,N-L node) for each P. In order to do that, we introduce the concept of *maximal continuation*:

Def: *Continuation and C(P,n) Function*
Let $n1$ and $n2$ be nodes in the hMSC$_G$ graph and P a process. $n2$ is a *continuation* of $n1$ for P if i) $n2$ may be reached from $n1$ in through one edge on the MSC$_G$ graph or if ii) there is a $n3$ such that $n3$ is a continuation of $n1$, $n2$ is a continuation of $n3$ and $n3$ does not contain events for process P.
The function C: P x $n \longrightarrow$ N identifies the continuation for process P in node n and returns a set of nodes N.

Informally, $n2$ is a continuation of $n1$, if $n2$ is directly reachable from $n1$ or it is reachable from $n1$ traversing nodes where P does not react.

Def: *Maximal Continuation and MC(P,n) Function*
Node $n2$ is a *maximal continuation* of $n1$ if i) $n2$ is a continuation of $n1$ and ii) for all $n3$ continuations of $n1$, $n2$ is a continuation of $n3$.
The function MC: P x $n \longrightarrow$ N identifies the maximal continuation for process P in node n and returns a set of nodes N.

Informally, $n2$ is a maximal continuation of $n1$ if $n2$ is the first node reachable from $n1$ so that P reacts. (Notice that the concept of continuation and maximal continuation just provided is more fine-grained of that provided in [21] since we deal with hMSC$_G$ nodes instead of hMSC nodes).

Using these definitions we can say that State(P,N-L node) = State(P,MC(P,N-L node)), assuming that State(P,N) = $\cup$ {State(P,n1), State(P,n2),..., State(P,ni)} where n1..ni $\in$ N. Informally, the state a process P has in the N-L node corresponds to the state P has in all the states reachable form N-L so that P can react.

As we said, this state is kept from P until its next reaction in the hMSC$_G$. Technically, State(P,ns) = State(P,N-L node), for each node ns so that $ns \in$ DR(N-L node) and P is not reacting in ns.

Moreover, to extend what we said in Sect. 4.1, if P is not reacting in the N-L node and N-$L \in$ C(P,np), then State(P,np) = State(P, N-L node). This happens because when a process does not react in a hMSC$_G$ node, he becomes ready to react as in the first next state it can react.

We can use the following algorithm to identify the state a generic process P has in the hMSG$_G$ nodes:

Algorithm to Identify State(P,n):

```
Build the hMSC_G graph for the MSC specification under analysis;
\\P is a generic process;
n = N-L node;
NS = {n1, n2, ..., nk};  \\ a set of nodes
FOR each n
    FOR each process P,
       compute MC(P,n) \\ in-breath visit of the graph
       State(P,n) = State(P,MC(P,n)) \\ P assumes the state of its
                                      \\maximal continuation
       NS = DR(n);
       \\ with this function we compute the state P has in the DR(n)
       FOR each P, compute NextState(P,NS);
       \\ with the following statements, we compute the state of P
        \\ in in the predecessors of the N-L node
       IF P does not react in n
         THEN
            FOR each node np so that n = MC(P,np),
                     State(P,np) = State(P,n).


NexState(P,NS)
    {FOR each node ns in NS
         IF State(P,ns) = empty
            THEN State(P,ns) = State(P,n);
                 NexState(P,DR(ns))}
```

Step 3: AB(P,n) Detection

Step 2 identified State(P,n), given a process P and a node n. As we said, it may happen that process P, after a N-L node and due to the non-local choice, can show some unexpected behaviors, called "augmented behavior". Those augmented behaviors are all those actions P can run in n but are not in its maximal continuation. The following definition may help to define this concept:

Def: *Augmented Behavior and AB(P,n) Function*
The augmented behavior of P in a node n is given by State(P,n) - State(P,MC(n)).
The function AB: P x $n \longrightarrow s$ identifies the augmented behavior for process P in node n and returns the augmented state s.

The following algorithm detects, for each process in each node n, the augmented behavior of P in that node.

Algorithm to Identify AB(P,n):

```
FOR each P in the MSC specification
    FOR each node n
        AB(P,n) = State(P,n) - State (P,MC(P,n)).

This information can be stored inside the hMSC_G graph.
```

Step 4: Implied Scenario identification
For each process P in each node n, we know what augmented behavior P has in n. If two processes P1 and P2 are augmented so that event e in AB(P1,n) is in AB(P2,n) too, then a node source of implied scenario is detected. A path that starts from the initial hMSC$_G$ graph node, traverses the N-L node and reaches n is implied if n is in C(Pi,N-L) for i= 1 or 2. The last condition requires that after the $N - L$ node at least one of the two processes do not react before the node source of the implied scenario.

The algorithm follows:

Algorithm to Identify Implied Scenarios Source:

```
n0 = first node in the hMSC_G graph;
p = a hMSC_G graph;
FOR each node n in the hMSC_G graph
    FOR each couple of processes P1 and P2
        IF exists "e" so that "e" is in AB(P1,n) and "e" is in AB(P2,n)
            THEN an implied scenario node is identified.
FOR each implied scenario source n,
    IF n is in C(P1,N-L) or n is in C(P2,N-L)
        THEN p = n0. ...  .N-L. ... .n is an implied scenario.
```

4.3 Application to the Boiler Example

The example we present and analyze in this section has been borrowed from [22]. The system is composed of four different processes communicating through four different MSCs, as shown in the MSC specification in Fig. 3a.

The system works in this way: after the "Control" process starts the system (in MSC *Initialise*), it acquires information about pressure (getP_Readings) and temperature (getT_Reading) from "Db1" and "Db2", respectively (in MSC *Analysis*). When the data have been received, fresh values are stored to the databases (MSC *Register*) and the system can nondeterministically proceed in three ways: the "Control" can get the new data (going back to MSC *Analysis*), the "Sensor" can refresh the data values in the databases (staying on MCS *Register*) or the "Control" can decide to Stop the system (in MSC *Stop*) and go back to the initial configuration (MSC *Initialise*). Figure 3b represents the hMSC$_G$ graph for this example that will be used to apply our approach.

Applying the Ben-Abdallah and Leue algorithm [2], we discovered that node 8 in Fig. 3b is a N-L node.

Applying the second step in our approach, we detect State(P,N-L node) for each process P. The output of this step is summarized in the next table:

Process	**State(P,N-L)**	**N-L ∪ (ns ∪ np)**
Control	{Stop, getP_Reading}	{8} + {6,7}
Sensor	{pressure, Stop }	{8} + {2,3,4,5,6}
Db1	{pressure, getP_Reading}	{8} + {1,9}
Db2	{temperature, getT_Reading}	{8} + {1,2,3,7,9}
Actuator	Command	{8} + {1,2,3,4,5,7,9}

The first column identifies the processes, the second proposes information on the state of each process in the N-L node and the last column shows the N-L and ns ∪ np nodes obtained applying Step 2 to the Boiler hMSC$_G$. The first row may be interpreted as: process Control in nodes 6, 7 and 8 can react with Stop and getP_Reading.

At this point, for each node $n \in N\text{-}L \bigcup \{ns\} \bigcup \{np\}$, Step 3 allows to identify the augmented behavior for each process P in n. The following table summarizes the results:

Process P	**node n**	**AB(P,n)**
Control	{6,7}	{Stop, getP_Reading}
Sensor	{2,3,4,5}	{pressure, Stop }
Sensor	{6}	Stop
Db1	{1}	pressure
Db1	{9}	{pressure,getP_Reading}
Db2	{1,2,8,9}	{temperature, getT_Reading}
Db2	{3}	{temperature}
Db2	{7}	{getT_Reading}
Actuator	{1,2,3,4,7,8,9}	Command

The first column represents the processes, the second one reports the nodes already found in the first table and the third column calculates the augmentation for P in node n. The first line can be read as follows: process Control in nodes 6 and 7 has some extra behaviors, namely Stop and getP_Reading.

Applying Step 4, we discover that Control and Sensor can communicate through an augmented action in node 6, i.e., Stop ∈ AB(Control,6) and Stop ∈ AB(Sensor,6). Path 1.2.3.4.5.6.7.8.2.3.4.5.6 in the hMSC$_G$ graph in Fig. 3b is implied since it starts from the first node, traverses node 8 (the $N - L$ node), reaches the implied scenario node 6 that is in C(Sensor,8).

4.4 Algorithm Completeness and Correctness

Completeness and correctness of the proposed approach are still under analysis. The approach we propose has been applied, by now, to many of the MSC

specifications available in [22] and several new specifications have been made in order to test completeness and correctness. In all the analyzed specifications, our algorithm has detected the same (and only the) implied scenarios detectable using the approach in [21] (that has been proved to be complete and correct).

An actual limitation (i.e., incompleteness) of our approach is that it works only with specifications composed by both MSCs and hMSCs. Therefore, the approach presented in [1] is more complete, since it can detect implied scenarios just starting from a set of MSCs. An idea to make our approach working without the hMSC specification is to build an MSC_G graph, identifying how the MSCs can interact, and to analyze this new graph with the proposed approach. This is just an initial idea that needs to be evaluated and extended in future work.

5 Related Work

There are some areas of research that may be related to our work: implied scenario detection, consistency checking and model checking.

Implied Scenarios:
The first research on implied scenarios have been proposed in [1,21]: the first paper proposes a polynomial-time algorithm to determine if a set of scenarios are "realizable" through state machines (i.e., if exists a concurrent automata which implement precisely those scenarios) and to synthesize such a realization using a deadlock-free model. If scenarios are not realizable, implied scenarios are detected. In the proposed approach, scenarios are modeled using Message Sequence Charts (MSCs) [12]. The former paper, based on both MSCs and hMSCs, describes an algorithm which generates some safety properties identifying (a simplification of) the exact behavior of the MSC specification, it models this properties using Labeled Transition Systems (LTSs) and checks the conformance of the synthesized LTS with respect to these properties. The algorithm has been implemented inside the Labeled Transition System Analyzer (LTSA) tool [22].

Consistency Checking:
As outlined in [4] consistency checking among multi-perspective or multi-language specifications is not a young field of research. There are different papers proposing different approaches to handle consistency. Some of these are used for checking model consistency.

In [5] the authors propose an approach for multiple views consistency checking. To check the views consistency, they provide a formal definition of the views and their relationships, they define their semantics as sets of graphs and apply an algorithm to check the consistency of diagrams. Filkenstein and colleagues in [4] describe how inconsistencies may be handled using the *ViewPoints* framework and a logic-based consistency handling.

Our work may be related to both of these papers since what we finally do is to check the conformance between a scenario and a state machine. The main

difference is that while in the other approaches these two models may be developed independently, in our case the state machine is obtained by synthesis from the scenario specification.

Model Checking:
Some model-checking based approaches have been proposed to model check scenarios with respect to state-based diagrams [18,6].

Again, these works have still in common with our the intent (i.e., checking the consistency between two different models) but in our case one model is obtained by synthesis from the other.

6 Comparison and Considerations

Comparison with the Approach in [21]:
In our approach, we propose a "structural" analysis which builds the hMSC_G graph and analyze its structure, in order to reveal implied scenarios. In [21], they build a behavioral model of the system, and analyze it. As a result, the complexity of our approach is not dependent on the system, behavioral, complexity, as opposite to [21].

To test the value of our approach, we modified the Boiler (version 3) specification by introducing two concurrently acting "Control" processes. The safety property, built by the LTSA-Implied tool [22] (and implementing the [21] approach) was modeled using an LTS composed by 748 nodes in the original specification and 11717 in the second one, almost 15 times bigger. The application of the approach we proposed, instead, requires to build a graph composed by 9 nodes in the first specification, 18 nodes in the second one. Moreover, the [21] approach needs to build the synthesized state machine, while we do not.

From this analysis, we cannot certainly conclude that our approach is always more efficient than that proposed in [21]. Anyway, we can argue that for concurrent and complex specifications, our approach can reduce time and memory (usage) complexity with respect to the approach presented in [21].

Another advantage of our approach is that with one execution *all* the implied scenarios in the hMSC specification can be discovered and displayed. On the opposite, the LTSA-Implied tool may discover only one implied scenario at a time providing a partial, incomplete, facet of the problem. With our approach, instead, we provide a software engineer a more valuable tool that detects all the possible sources of MSC underspecification at the same time and allows him to fix all the implied behaviors through only one specification refinement.

Implied Scenarios and Non-local Branching Choices:
If an implied scenario is detected, a non-local branching choice (NLC) holds. The opposite is not true, as already pointed out in Sect. 3; an NLC not always causes an implied scenario. In general, it could be enough to detect the NLC and fix it (without using the algorithm proposed in this paper). However, the application of the proposed algorithm guarantees some important advantages.

Every time a NLC is detected and fixed, new problems can be introduced in the MSC specification. Using our algorithm, we can decide to fix only those NLCs that give rise to an implied scenario. Another interesting point is that applying the described algorithm it becomes easier to *localize and fix* the implied scenarios (when they are not desired). In fact, the algorithm we propose uses the algorithm proposed in [2] to identify the non-local choice node(s), discovers how the processes behavior has been augmented due to the non-local choice and outlines which processes have to communicate to generate the implied scenario.

The Message Flow Graph (MFG) is used in [2] and implemented inside the MESA tool [3] in order to detect NLCs. Since the MFG seems to capture more information than the hMSC_G (used in our approach to detect implied scenarios), a MFG could be used to detect both NLCs and implied scenarios. Further research is necessary to understand how our algorithm needs to be modified to work with the MFGs. Eventually, our algorithm could be integrated in the MESA tool in order to provide a complete structural approach to detect both NLCs and implied scenarios.

7 Future Work

Future work goes along different directions: first of all, we need to analyze the completeness of the approach and how it can be extended to cover specification composed of MSCs only. A tool needs to be realized in order to implement the algorithm. A possible alternative may integrate the approach in existing tools, like MESA [3]. Future research will be conducted to analyze the applicability of this approach to UML [15] diagrams: the semantic differences in between sequence diagrams and MSCs and the absence of mechanisms to combine sequence diagrams together need to be carefully analyzed. An interesting future work could also analyze how these unexpected, implied scenarios can be used for testing purposes. In the meantime, we are looking for a real system specification to better evaluate the qualities/weaknesses of the proposed approach.

Acknowledgments. The author would like to acknowledge the University of California Irvine and the Italian M.I.U.R. National Project SAHARA that supported this work, Paola Inverardi for her comments on a draft of this paper and the anonymous reviewers for their interesting suggestions.

References

1. R. Alur, K. Etessami, and M. Yannakakis. Inference of Message Sequence Charts. In Proc. *22nd Int. Conf. on Software Engineering*, ICSE2000, Limerick, Ireland.
2. H. Ben-Abdallah and S. Leue. Syntactic Detection of Process Divergence and non-Local Choice in Message Sequence Charts. In Proc. *TACAS'97*, LNCS 1217, pp. 259–274, 1997.
3. H. Ben-Abdallah and S. Leue. MESA: Support for Scenario-Based Design of Concurrent Systems. In Proc. *TACAS'98*, LNCS 1384, pp. 118–135, 1998.

4. A. Filkenstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specifications. *IEEE Trans. on Software Engineering*, Vol. 20, Number 8, pp. 569–578, August 1994.
5. P. Fradet, D. Le Metayer, and M. Perin. Consistency Checking for Multiple View Software Architectures. In Proc. European Software Engineering Conference (*ESEC/FSE'99*), pp. 410–428, 1999.
6. P. Inverardi, H. Muccini, and P. Pelliccione. Automated Check of Architectural Models Consistency using SPIN. In *ACM Proc. Int. Conference on Automated Software Engineering* (ASE 2001), 2001.
7. K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software Practice and Experience*, 24(7), pp. 643–658, 1994.
8. K. Koskimies, T. Männistö, T. Systä, and J. Tuomi. Automated support for OO software. *IEEE Software*, 15(1), pp. 87–94, 1998. Tool Download at: `http://www.cs.tut.fi/~tsysta/sced/`.
9. I. Krüger, R. Grosu, P. Scholz, and M. Broy. From MSCs to Statecharts. *Distributed and Parallel Embedded Systems*, Kluwer Academic, 1999.
10. S. Leue, L. Mehrmann, and M.Rezai. Synthesizing ROOM models from message sequence chart specifications. In *Proc. of the 13th IEEE Conf. on Automated Software Engineering*, 1998.
11. S. Mauw, M.A. Reniers, and T.A.C. Willemse. Message Sequence Charts in the Software Engineering Process. In S.K. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., Vol. 1, *Fundamentals*, pp. 437–463, December 2001.
12. Message Sequence Chart (MSC). ITU Telecommunication Standardization Sector (ITU-T). Z.120 Reccomendation for MSC-2000, year 2000.
13. H. Muccini. An Approach for Detecting Implied Scenarios. In Proc. *ICSE2002 Workshop on "Scenarios and state machines: models, algorithms, and tools"*. Available at `http://www.henrymuccini.com/publications.htm`.
14. C. Pons, R. Giandini, and G. Baum. Dependency Relations Between Models in the Unified Process. In Proc. *IWSSD 2000*, November 2000.
15. Rational Corporation. Uml Resource Center. UML documentation, version 1.3. Available from: `http://www.rational.com/uml/index.jtmpl`.
16. G. Reggio, M. Cerioli, and E. Astesiano. Towards a Rigorous Semantics of UML Supporting its Multiview Approach. In Proc. *FASE 2001*, LNCS n. 2029, Berlin, Springer Verlag, 2001.
17. E. Rudolph, P. Graubmann, and J. Grabowski. Tutorial on Message Sequence Charts. In *Computer Network and ISDN Systems*, 28(12), pp. 1629–1641, 1996. Special issue on SDL and MSC.
18. T. Schafer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. On the *Workshop on Software Model Checking*, Paris, July 23 2001. ENTCS, volume 55 number 3.
19. Second International Workshop on Living with Inconsistency. ICSE'01 workshop, May 13, 2001 Toronto, Canada.
20. UBET. `http://cm.bell-labs.com/cm/cs/what/ubet/`.
21. S. Uchitel, J. Kramer, and J. Magee. Detecting Implied Scenarios in Message Sequence Chart Specifications. In Proc. European Software Engineering Conference (*ESEC/FSE'01*), Vienna 2001.
22. S. Uchitel, J. Magee, and J. Kramer. LTSA and implied Scenarios. On line at: `http://www.doc.ic.ac.uk/~su2/Synthesis/`.
23. J. Whittle and J. Schumann. Generating Statechart Designs from Scenarios. In *Proc. 22nd Int. Conference on Software Engineering* (ICSE'00), 2000.

Capturing Overlapping, Triggered, and Preemptive Collaborations Using MSCs

Ingolf H. Krüger

Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA, 92093-0114, USA
ikrueger@ucsd.edu

Abstract. Message Sequence Charts (MSCs) and related notations have found wide acceptance for scenario-oriented behavior specifications. However, MSCs lack adequate support for important aspects of interaction modeling, including overlapping interactions, progress/liveness specifications, and preemption. Such support is needed particularly in the context of *service-oriented* specifications both in the business information and embedded systems domain. In this text, we introduce extensions to the "standard" MSC notation addressing these deficits, and provide a semantic foundation for these extensions.

1 Introduction

The complexity of software-enabled systems continues to rise. Driven by the wired and wireless incarnations of the Internet, the networking of formerly monolithic devices and their software components increases. More and more systems emerge as a collaboration between peer networking nodes, each offering software functions to its environment, and utilizing the functions offered by others.

As a consequence, the focus of concern in requirements capture, design, and deployment of software solutions shifts from individual computation nodes to their interaction. This shift of concern is exemplified by the current trend towards "web services". Here, software functions (called *services*) are published as individual entities at well-known addresses on the Internet; these functions can be consumed by others, yielding possibly complex, composite services.

To model and implement such interaction-based services systematically, expressive description techniques and methodological foundation for component interaction are essential.

Typical software development approaches and modeling languages, however, place their focus on the construction of individual software components, instead of on component collaboration. The Unified Modeling Language (UML)[16] is a typical example (at least before version 2.0, which is still under discussion). Its syntactic means – and corresponding tool support – for specifying state-based behavior of individual components (statechart diagrams) are far better developed than the corresponding notations for interaction patterns (activity, sequence and collaboration diagrams).

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 387–402, 2003.

Message Sequence Charts (MSCs)[7,8], on the other hand, have been widely accepted as a valuable means of visualizing and specifying asynchronous component interaction. Their potentials in this regard have earned them entrance into the current suggestion for the UML's 2.0 standard, where they are adapted to support modeling of a wide range of communication concepts.

Both of these notations, however, provide only very basic support for interaction specifications. In this text, we show how to extend the expressiveness of MSCs and sequence diagrams to include:

- Overlapping interaction patterns
- Liveness/progress properties
- Preemption specifications.

In the following paragraphs we describe each of these extensions in more detail. Although there are many other areas for improvement we could address – including proper handling of data in sequence diagrams, and hierarchical refinement for messages and components, to mention just two examples – for reasons of brevity we focus on the three extensions listed below.

1.1 Overlapping Interaction Patterns

We call sequences of interactions in which some communication partners and some of the messages they exchange coincide *overlapping*; overlapping interaction patterns are extremely important in service-oriented specifications. Each individual service only represents a *partial* view on the collaborations within the system under consideration. To get the *overall* picture for one implementation component, say, all the different services in which a single component is involved need to be joined. This requires an adequate composition operator for making the relationship between different service specifications precise.

1.2 Trigger Composition

Similarly important is availability of an operator for specifying liveness/progress in MSCs. Most sequence diagram dialects provide means for indicating alternative interaction patterns; they fail, however, to offer notation for indicating which alternatives should be selected to make progress towards a desirable goal. Consequently, liveness properties can only be described as a side remark or in another modeling language, such as an state-automaton-based approach. Without proper support for liveness, sequence diagrams cannot mature beyond scenario specifications. Specifically, we are interested in working with abstract progress properties, such as "if a certain interaction pattern has occurred in the system, then another one is inevitable", or "one interaction pattern *triggers* another".

1.3 Preemption

Preemption is a fundamental concept especially in technical and embedded systems development. Although the very roots of MSCs are in telecommunication

systems, where preemption scenarios abound – think of specifying a telephone call, where at any time either party can hang up –, no support for preemption exists in either MSCs or sequence diagrams.

1.4 Contributions and Outline

Our contributions in this text are twofold. First, in Sect. 2, we provide a motivating "toy" example, illustrating the usability and applicability of the concepts just outlined. This example serves also to introduce the extensions to the MSC syntax we use, and how they integrate with the "standard" notation. Although we have chosen to stay close to the MSC-96 syntax, the example illustrates that our suggestions could also be transferred to other interaction-based specification techniques, such as the UML's sequence diagrams.

Second, in Sect. 3 we provide a formal semantics for MSCs including support for the new composition operators that address overlapping scenarios, triggered and preemptive collaborations. The basis for this semantics is a precise system model for component interaction, based on streams. There have certainly been other attempts at semantics definition for MSCs before; the combination of explicit concepts for overlapping, trigger composition, and preemption of MSCs is, however, not treated consequently in these approaches. In addition, our approach has the advantage of supporting systematic MSC refinement as well as component synthesis from MSCs – for reasons of brevity we have to refer the reader to [9] for more details on these issues.

We discuss related work in Sect. 4, as well as our conclusions and future work in Sect. 5.

2 Example: The Abracadabra-Protocol

To illustrate the applicability of the suggested operators, we model a simplified version of the Abracadabra communication protocol[1,2] using MSCs[7,15,9]. To describe this protocol we assume given a system consisting of two distinct components X and Y; we assume further that these two components communicate via messages sent along channels xy (from X to Y), and yx (from Y to X). Figure 1 shows this component structure in graphical form.

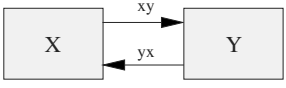


Fig. 1. System Structure Diagram (SSD) for the Abracadabra-protocol

The symmetric Abracadabra-protocol describes a scheme that allows any of the two components to establish a connection to the other component, send data messages once a connection exists, and tear down an existing connection it

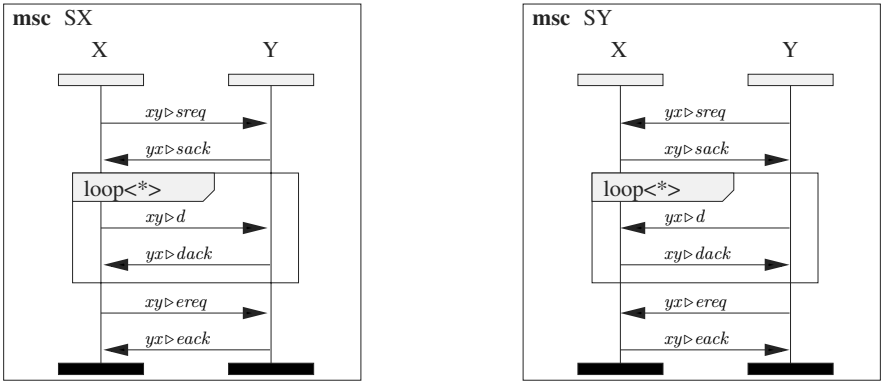
has initiated. If both components try to establish a connection simultaneously, the system is in conflict. Then, both components tear down their "attempted" connections to resolve the conflict.

Section 3 contains the formal definitions corresponding to the concepts and operators introduced informally here.

2.1 Message Sequence Charts

MSCs provide a rich graphical notation for capturing interaction patterns. MSCs have emerged in the context of SDL[6] as a means for specifying communication protocols in telecommunication systems. They have also found their way into the new UML 2.0 standard[14], which significantly improves the role of interaction models within the UML.

MSCs come in two flavors: Basic and High-Level MSCs (HMSCs). A basic MSC consists of a set of axes, each labeled with the name of a component. An axis represents a certain segment of the behavior displayed by its corresponding component. Arrows in basic MSCs denote communication. An arrow starts at the axis of the sender; the axis at which the head of the arrow ends designates the recipient. Intuitively, the order in which the arrows occur (from top to bottom) within an MSC defines possible sequences of interactions among the depicted components.



(a) (b)

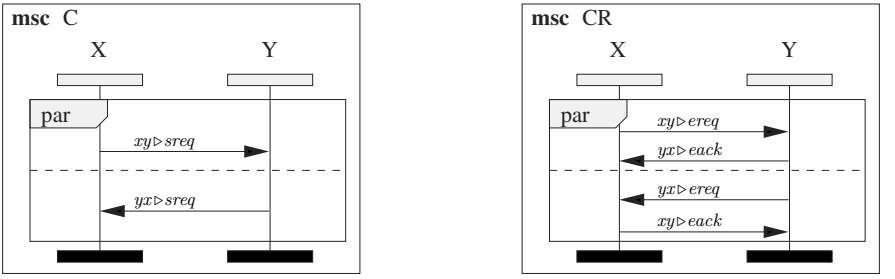
Fig. 2. MSCs for successful transmission

As an example, consider the MSC of Fig. 2a. It depicts how X and Y interact to establish successful data transmission. X initiates the interaction by sending message $xy!sreq$ ("sending requested") to Y. Upon receipt of Y's reply $yx!sack$ ("sending acknowledged"), X sends an arbitrary, finite number of $xy!d$ messages. Each data message is acknowledged individually by Y; X waits for a $yx!dack$ message from Y before sending the next data message. Graphically,

repetition is indicated by the loop box enclosing the recurring messages. To close the transmission X sends message $xy!ereq$ ("end requested") to Y; Y acknowledges transmission termination by means of a $yx!eack$ ("end acknowledged") message. Figure 2b shows the symmetric case, where Y is the initiator.

Syntactically, we have adopted a slightly modified version of MSC-96[7]; our message arrows carry an indicator for the channel on which a message is sent in addition to the message itself; we will come back to this in Sect. 3.2. Moreover, we use unbounded loops, which are not available in MSC-96.

Conflict in the ABRACADABRA protocol occurs if both X and Y try to establish a connection simultaneously. The MSC in Fig. 3a captures this case by means of causally unrelated messages, using the "parallel box" syntax of MSC-96. Conflict resolution is handled by mutual exchange of messages *ereq* and *eack*



(a) (b)

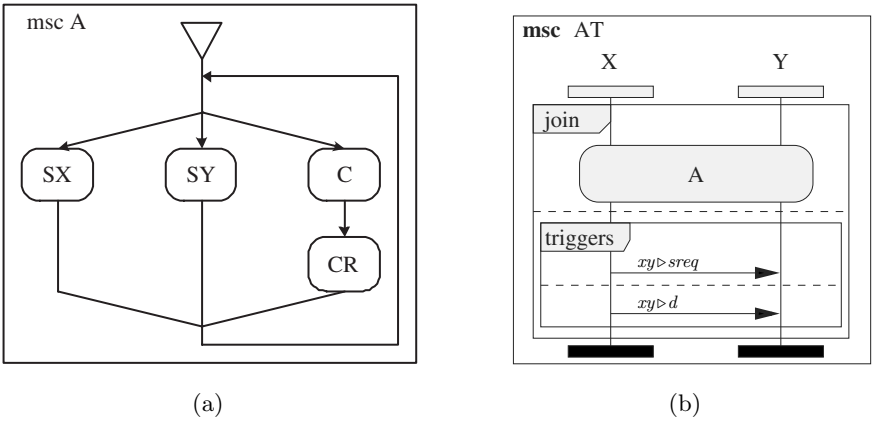
Fig. 3. MSCs for conflicts and their resolution

by X and Y. Again, there need not be a specific order between the *ereq* and *eack* messages with different origins (cf. Fig. 3b).

An HMSC is a graph whose nodes are references to other (H)MSCs. The semantics of an HMSC is obtained by following paths through the graph and composing the interaction patterns referred to in the nodes along the way. The HMSC of Fig. 4a, for instance, specifies that every system execution is an infinite sequence of steps, where each step's behavior is described by one of the following: MSC SX (successful communication initiated by X), MSC SY (successful communication initiated by Y), or MSC C (conflict) followed by MSC CR (conflict resolution). Multiple vertices emanating from a single node in the HMSC graph indicate nondeterministic choice.

2.2 Introducing Progress/Liveness

So far, we have left open whether a send request by either component will, eventually, result in an established connection. The use of nondeterministic choice in the definition of MSC A allows an infinite sequence of steps consisting only of conflict and conflict resolution.



(a) (b)

Fig. 4. HMSC for the ABRACADABRA-protocol, and ABRACADABRA with progress property

We introduce a new composition operator, called "trigger composition", to cast the progress/liveness property (cf. [4,13]) that a message $xy?sreq$ must lead to subsequent data exchange, i.e. occurrence of at least one $xy?d$ message.

Informally speaking, we write $\square \mapsto \square$ to indicate that whenever the interaction pattern specified by MSC $\square$ has occurred in the system under consideration, it is eventually followed by an occurrence of the interaction pattern specified by MSC $\square$.

Having trigger composition available, we can describe the progress property mentioned above as $xy?sreq \mapsto xy?d$; see Fig. 4b for the graphical syntax we use for trigger composition.

2.3 Joining Overlapping MSCs

By now, we have two separate MSCs describing system behaviors. On the one hand we have MSC A, which describes the major interaction patterns of the ABRACADABRA protocol. On the other hand we have the MSC $xy?sreq \mapsto xy?d$, which describes a progress property relating occurrences of messages $xy?sreq$ and $xy?d$.

Our next step is to compose these two MSCs such that the resulting MSC contains only paths through the ABRACADABRA protocol that fulfill the progress property. To that end, we introduce the "join" composition operator for MSCs. The join $\square \otimes \square$ of two MSCs $\square$ and $\square$ describes behaviors complying to both MSCs such that identical messages occurring in both MSCs are identified.

The join operator in Fig. 4b "binds" the messages occurring in the trigger composition $xy?sreq \mapsto xy?d$ to those in A. The semantics of the joint MSC is the subset of A's semantics where every $xy?sreq$ message is followed by an $xy?d$ message eventually. Put another way, no element of the semantics of the joint

MSC has only conflicts or successful transmissions initiated by Y, if X issues message $xy!sreq$ at least once.

2.4 Introducing Preemption

To demonstrate an application of preemption we extend the informal protocol specification given above as follows: we modify the system structure from Fig. 1 by connecting component X to the "environment" (represented by component *ENV*) through channel ex (cf. Fig. 5).

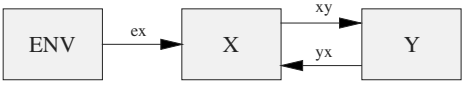
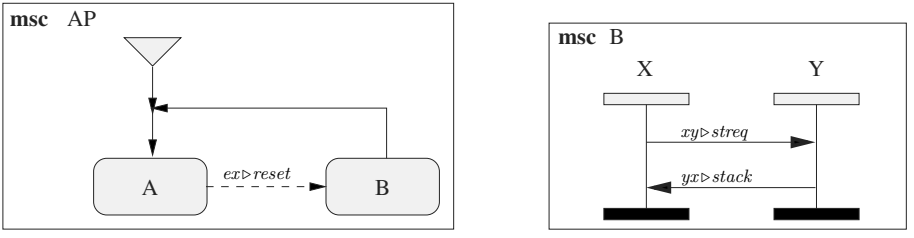


Fig. 5. SSD for the ABRACADABRA-protocol with preemption

By sending message *reset* along channel ex the environment can force X to stop any communication it may currently be involved in, and to restart the whole protocol afresh. Upon receipt of message *reset*, component X sends message *streq* ("stop requested") to Y; Y replies by sending message *stack* ("stop acknowledged") to X.

The HMSC AP from Fig. 6a models this behavior by means of a preemption arrow labeled with the preemptive message $ex!reset$. MSC B (cf. Fig. 6b) shows the handling of the preemption. Recall that MSC A (cf. Fig. 4a) captures the overall behavior of the ABRACADABRA-protocol (without preemption).



(a) (b)

Fig. 6. Preemption and its handling

Without the preemption construct we would have to rewrite the MSCs A, SX, SY, C, and CR completely to accommodate the external reset request. The resulting MSCs would lose their intuitive appeal almost entirely, because the one exceptional case would dominate the whole specification.

3 A Formal Framework for Precise MSC Specifications

In this section we introduce the formal framework for the semantics definition of MSCs. We use this framework, in particular, to describe the semantics of the operators for the join, trigger, and preemptive composition of MSCs.

3.1 System Model

We prepare our precise semantics definition for MSCs by first introducing the structural and behavioral model (the *system model*) on which we base our work. Along the way we introduce the notation and concepts we need to describe the model.

System Structure. Structurally, a system consists of a set P of components, objects, or processes[1], and a set C of named channels. Each channel $ch \in C$ is directed from its source to its destination component; we assume that channel names are unique. Channels connect components that communicate with one another; they also connect components with the environment. Communication proceeds by message exchange over these channels.

With every $p \in P$ we associate a unique set of states, i.e. a component state space, S_p. We define the state space of the system as $S \stackrel{\mathsf{def}}{=} \square_{p \in P} S_p$. For simplicity, we represent messages by the set M of message identifiers.

System Behavior. Now we turn to the dynamic aspects of the system model. We assume that the system components communicate among each other and with the environment by exchanging messages over channels. We assume further that a discrete global clock drives the system. We model this clock by the set $\mathbb{N}$ of natural numbers. Intuitively, at time $t \in \mathbb{N}$ every component determines its output based on the messages it has received until time $t - 1$, and on its current state. It then writes the output to the corresponding output channels and changes state. The delay of at least one time unit models the processing time between an input and the output it triggers; more precisely, the delay establishes a strict causality between an output and its triggering input (cf. [3,2]).

Formally, with every channel $c \in C$ we associate the histories obtained from collecting all messages sent along c in the order of their occurrence. Our basic assumption here is that communication happens asynchronously: the sender of a message does not have to wait for the latter's receipt by the destination component.

This allows us to model channel histories by means of *streams*. Streams and relations on streams are an extremely powerful specification mechanism for distributed, interactive systems (cf. [3,17]). Here, we only use and introduce a small

[1] In the remainder of this document, we use the terms components, objects, and processes interchangeably.

fraction of this rich semantic model; for a thorough introduction to the topic, we refer the reader to [17,3].

A stream is a finite or infinite sequence of messages. By X^* and X^∞ we denote the set of finite and infinite sequences over set X, respectively. $X^\Box \stackrel{\text{def}}{=} X^* \cup X^\infty$ denotes the set of streams over set X. We identify X^* and X^∞ with $\bigcup_{i\in\mathbb{N}}([0..i] \to X)$ and $\mathbb{N} \to X$, respectively, and use function application to write $x.n$ for the n-th element of stream x (for $x \in X^\Box$ and $n \in \mathbb{N}$).

We define $\tilde{C} \stackrel{\text{def}}{=} C \to M^*$ as a channel valuation that assigns a sequence of messages to each channel; we obtain the timed stream tuple $\tilde{C}^\infty$ as an infinite valuation of all channels. This models that at each point in time a component can send multiple messages on a single channel.

With timed streams over message sequences we have a model for the communication among components over time. Similarly we can define a succession of system states over time as an element of set S^∞.

With these preliminaries in place, we can now define the semantics of a system with channel set C, state space S, and message set M as an element of $\mathcal{P}((\tilde{C} \times S)^\infty)$. For notational convenience we denote for $\Box \in (\tilde{C} \times S)^\infty$ by $\Box_1(\Box)$ and $\Box_2(\Box)$ the projection of $\Box$ onto the corresponding infinite channel and state valuations, respectively; thus, we have $\Box_1(\Box) \in \tilde{C}^\infty$ and $\Box_2(\Box) \in S^\infty$. The existence of more than one element in the semantics of a system indicates nondeterminism.

3.2 MSC Semantics

In the following we establish a semantic mapping from MSCs to the formal framework introduced above. In the interest of space we constrain ourselves to a significant subset of the notational elements contained in MSC-96 and UML 2.0, yet provide the extensions for overlapping scenarios (join operator), progress (trigger composition), and preemption. For a comprehensive treatment of MSC syntax and semantics we refer the reader to [9]; further approaches to defining MSC semantics are discussed in Sect. 4.

Preliminaries. To facilitate the semantics definition we use a simplified textual syntax for MSCs. The base constructors for MSCs are **empty**, $c!m$ and **any**, denoting the absence of interaction (empty MSC), the sending of message m on channel c, and arbitrary interactions, respectively. Given two MSCs $\Box$ and $\Box$ we denote by $\Box\ ;\ \Box$ and $\Box\ \sim\ \Box$ the sequencing and interleaving of $\Box$'s and $\Box$'s interaction patterns, respectively. If g represents a predicate on the state space of the system under consideration, then we call $g : \Box$ a guarded MSC; intuitively it equals **empty** if g evaluates to false, and $\Box$ otherwise. By $\Box \otimes \Box$ we denote the join of MSCs $\Box$ and $\Box$. The join of two MSCs corresponds to the interleaving of the interaction sequences they represent with the exception that common messages on common channels synchronize. The trigger composition, written $\Box \mapsto \Box$ of two MSCs expresses the property that whenever the interactions specified by $\Box$ have occurred the interactions specified by $\Box$ are inevitable. We write $\Box \stackrel{ch!m}{\rightarrow} \Box$

to denote preemption. If message $ch!m$ occurs during the behavior represented by $\square$, then this behavior is preempted and continued by the behavior that $\square$ represents. Intuitively, we can think of $\square$ as the preemption/exception handler, and of $ch!m$ as the exception being thrown. To express the restarting of $\square$ upon occurrence of preemptive message $ch!m$, we write $\square\Uparrow_{ch!m}$. $\square\uparrow_g$ denotes a "while" loop, repeating the interactions of $\square$ while g evaluates to true; a special case is $\square\uparrow_\infty$, which denotes an infinite repetition of $\square$. We define $\square^0 \stackrel{\text{def}}{=} \textbf{empty}$, and $\square^{i+1} \stackrel{\text{def}}{=} (\square\ ;\ \square^i)$ for $i \in \mathbb{N}$. $\square\uparrow_*$ denotes unbounded finite repetition of $\square$.

An MSC definition associates a name with an interaction specification, written **msc** X = $\square$. By ⟨MSC⟩ and ⟨MSCNAME⟩ we denote the set of all syntactically correct MSCs, and MSC names, respectively. An MSC document consists of a set of MSC definitions (assuming unique names for MSCs within a document). To reference one MSC from within another we use the syntax $\rightarrow Y$, where Y is the name of the MSC to be referenced.

Example: As an example for the representation of MSCs in the syntax introduced above we consider again the service depicted in Fig. 2a. In our textual syntax the scenario is expressed as follows:

$$\textbf{msc}\ \mathrm{SX} =$$
$$xy!sreq\ ;\ yx!sack\ ;\ (xy!d\ ;\ yx!d)\uparrow_*\ ;\ xy!ereq\ ;\ yx!eack$$

The textual MSC definition corresponding to the graphical representation in Fig. 6a is

$$\textbf{msc}\ AP = ((\rightarrow A) \stackrel{ex!reset}{\rightarrow} (\rightarrow B))\uparrow_{<\infty>}$$

Denotational Semantics. In this section we introduce the semantic mapping from the textual representation of MSCs into the semantic domain $(\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. Intuitively, we associate with a given MSC a set of channel and state valuations, i.e. a set of system behaviors according to the system model we have introduced in Sect. 3.1. Put another way, we interpret an MSC as a constraint on the possible behaviors of the system under consideration. More precisely, with every $\square \in \langle\mathrm{MSC}\rangle$ and every $u \in \mathbb{N}_\infty$ we associate a set $[\![\square]\!]_u \in \mathcal{P}((\tilde{C}\times S)^\infty \times \mathbb{N}_\infty)$; any element of $[\![\square]\!]_u$ is a pair of the form $(\square \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$. The first constituent, $\square$, of such a pair describes an infinite system behavior. u and the pair's second constituent, t, describe the time interval within which $\square$ constrains the system's behavior. Intuitively, u corresponds to the "starting time" of the behavior represented by the MSC; t indicates the time point when this behavior has finished. Hence, outside the time interval specified by u and t the MSC $\square$ makes no statement whatsoever about the interactions and state changes happening in the system. To model that we cannot observe (or constrain) system behavior "beyond infinity" we define that for all $\square \in (\tilde{C} \times S)^\infty$, $\square \in \langle\mathrm{MSC}\rangle$, and $t \in \mathbb{N}_\infty$ the following predicate holds: $(\square \lhd t) \in [\![\square]\!]_\infty$.

We assume given a relation $MSCR \subseteq \langle\mathrm{MSCNAME}\rangle \times \langle\mathrm{MSC}\rangle$, which associates MSC names with their interaction descriptions. We expect $MSCR$ to be

the result of parsing all of a given MSC document's MSC definitions. For every MSC definition **msc** $X = \square$ in the MSC document we assume the existence of an entry $(X \lhd \square)$ in *MSCR*. For simplicity we require the MSC term associated with an MSC name via *MSCR* to be unique.

Empty MSC. For any time $u \in \mathbb{N}_\infty$ **empty** describes arbitrary system behavior that starts and ends at time u. Formally, we define the semantics of **empty** as follows:

$$[\![\mathbf{empty}]\!]_u \stackrel{\mathsf{def}}{=} \{(\square \lhd u) : \square \in (\tilde{C} \times S)^\infty\}$$

Arbitrary Interactions. MSC **any** describes completely arbitrary system behavior; there is neither a constraint on the allowed interactions and state changes, nor a bound on the time until the system displays arbitrary behavior:

$$[\![\mathbf{any}]\!]_u \stackrel{\mathsf{def}}{=} \{(\square \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : t \geq u\}$$

any has no direct graphical representation; we use it to resolve unbound MSC references (see below).

Single Message. An MSC that represents the occurrence of message m on channel ch constrains the system behavior until the minimum time such that this occurrence has happened:

$$[\![ch \square m]\!]_u \stackrel{\mathsf{def}}{=} \{(\square \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N} : t = \min\{v : v > u \wedge m \in \square_1(\square) \bowtie v \bowtie ch\}\}$$

Because we disallow pairs $(\square \lhd \infty)$ in $[\![ch \square m]\!]_u$ we require the message to occur eventually (within finite time). This corresponds with the typical intuition we associate with MSCs: the depicted messages do occur within finite time.

We add the channel identifier explicitly to the label of a message arrow in the graphical representation; this is useful in situations where a component has more than one communication path to another component.

Sequential Composition. The semantics of the semicolon operator is sequential composition (*strong sequencing* in the terms of [7]): given two MSCs $\square$ and $\square$ the MSC $\square$; $\square$ denotes that we can separate each system behavior in a prefix and a suffix such that $\square$ describes the prefix and $\square$ describes the suffix:

$$[\![\square \,;\, \square]\!]_u \stackrel{\mathsf{def}}{=} \{(\square \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \langle \exists t' \in \mathbb{N}_\infty :: (\square \lhd t') \in [\![\square]\!]_u \ \wedge \ (\square \lhd t) \in [\![\square]\!]_{t'} \rangle\}$$

Guarded MSC. Let $K \subseteq P$ be a set of instance identifiers. By p_K we denote a predicate over the state spaces of the instances in K. Let $[\![p_K]\!] \in \mathcal{P}(S)$ denote the set of states in which p_K holds. Then we define the semantics of the guarded MSC $p_K : \square$ as the set of behaviors whose state projection fulfills p_K at time u, and whose interactions proceed as described by MSC $\square$:

$$[\![p_K : \square]\!]_u \stackrel{\text{def}}{=} \{(\square \vartriangleleft t) \in [\![\square]\!]_u : \square_2(\square) \bowtie u \in [\![p_K]\!]\}$$

We require p_K to hold only at instant u. This allows arbitrary state changes from time u on. In particular, at no other point within the time interval covered by $\square$ can we assume that p_K still holds.

Alternative. An alternative denotes the union of the semantics of its two operand MSCs. The operands must be guarded MSCs; the disjunction of their guards must yield true. Thus, for $\square = p : \square'$, $\square = q : \square'$ with $\square' \vartriangleleft \square' \in \langle \text{MSC} \rangle$, and guards $p \vartriangleleft q$ with $p \vee q \equiv \text{true}$ we define:

$$[\![\square \mid \square]\!]_u \stackrel{\text{def}}{=} [\![\square]\!]_u \cup [\![\square]\!]_u$$

For guards p and q with $p \wedge q \equiv \text{true}$ the alternative expresses a nondeterministic choice.

References. If an MSC named X exists in the given MSC document, i.e. there exists a pair $(X \vartriangleleft \square) \in \mathit{MSCR}$ for some $\square \in \langle \text{MSC} \rangle$, then the semantics of a reference to X equals the semantics of $\square$. Otherwise, i.e. if no adequate MSC definition exists, we associate the meaning of **any** with the reference:

$$[\![\rightarrow X]\!]_u \stackrel{\text{def}}{=} \begin{cases} [\![\square]\!]_u & \text{if } (X \vartriangleleft \square) \in \mathit{MSCR} \\ [\![\textbf{any}]\!]_u & \text{else} \end{cases}$$

To identify **any** with an unbound reference has the advantage that we can understand the binding of references as a form of property refinement (cf. [9]).

Interleaving. Intuitively, the semantics of interleaving MSCs $\square$ and $\square$ "merges" elements $(\square \vartriangleleft t) \in [\![\square]\!]_u$ with elements $(\square \vartriangleleft t') \in [\![\square]\!]_u$. Formal modeling of this merge is straightforward, albeit more technically involved; we refer the reader to [9] for the details.

Join. The join $\square \otimes \square$ of two operand MSCs $\square$ and $\square$ is similar to their interleaving with the exception that the join identifies common messages, i.e. messages on the same channels with identical labels in both operands. MSC-96 does not offer an operator with a similar semantics.

$$\begin{gathered}
[\![\square \otimes \square]\!]_u \stackrel{\text{def}}{=} \{(\square \vartriangleleft t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
\langle \exists t_1 \vartriangleleft t_2 :: (\square \vartriangleleft t_1) \in [\![\square]\!]_u \wedge (\square \vartriangleleft t_2) \in [\![\square]\!]_u \wedge t = \max(t_1 \vartriangleleft t_2) \rangle \\
\wedge \langle \forall X \in (\mathit{msgs}\square \cap \mathit{msgs}\square)^* \vartriangleleft \square \in (\tilde{C} \times S)^\infty \vartriangleleft \mathit{ch} \in C \vartriangleleft t' \in [u \vartriangleleft t] \cap \mathbb{N} :: \\
((X \neq \langle\rangle) \wedge (\square_1(\square) \bowtie t' \triangleright \mathit{ch} = \square_1(\square) \bowtie t' \triangleright \mathit{ch} \setminus X)) \\
\Rightarrow \langle \forall t'' \in \mathbb{N} :: (\square \vartriangleleft t'') \not\in [\![\square]\!]_u \wedge (\square \vartriangleleft t'') \not\in [\![\square]\!]_u \rangle \rangle\}
\end{gathered}$$

In this definition we use the notation $m \backslash n$ as a shorthand for the stream obtained from m by dropping all elements that do also appear in n. By $msgs(\mathcal{M})$ we denote the set of message labels occurring in MSC $\mathcal{M}$.

The second outer conjunct of this definition ensures that we *cannot* reconstruct the behaviors of $\mathcal{M}_1$ and $\mathcal{M}_2$ independently from the behaviors of their join, if the two MSCs have messages in common. This distinguishes the join clearly from the interleaving of $\mathcal{M}_1$ and $\mathcal{M}_2$, if $msgs(\mathcal{M}_1) \cap msgs(\mathcal{M}_2) \neq \emptyset$ holds. In this state of affairs, we call $\mathcal{M}_1$ and $\mathcal{M}_2$ *non-orthogonal* (or *overlapping*); if $msgs(\mathcal{M}_1) \cap msgs(\mathcal{M}_2) = \emptyset$ holds, then we call $\mathcal{M}_1$ and $\mathcal{M}_2$ *orthogonal* (or *non-overlapping*).

The definition of the join operator is quite restrictive; for example, consider the two MSCs $\mathcal{M}_1 = c!m\,;\ c!n$ and $\mathcal{M}_2 = c!n\,;\ c!m$, which define two different orderings of the messages $c!m$ and $c!n$. It is easy to see that we have $[\![\mathcal{M}_1 \otimes \mathcal{M}_2]\!]_u = \emptyset$.

Preemption. The semantics of MSC $\mathcal{M} \overset{ch!m}{\rightarrow} \mathcal{N}$ is equivalent to the one of $\mathcal{M}$ as long as message $ch!m$ has not occurred. From the moment in time at which $ch!m$ occurs, the MSC immediately switches its semantics to the one given by $\mathcal{N}$:

$$
\begin{aligned}
[\![\mathcal{M} \overset{ch!m}{\rightarrow} \mathcal{N}]\!]_u \overset{\text{def}}{=}\ & \{(\mathcal{M} \lhd t) \in [\![\mathcal{M}]\!]_u : \langle \forall v \in \mathbb{N} : u \leq v \leq t : m \not\in \mathcal{M}_1(\mathcal{M}) \rhd v \rhd ch \rangle\} \\
& \cup \{(\mathcal{M} \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
& \quad \langle \exists v : v \in \mathbb{N} : \\
& \qquad v = \min\{t' : t' > u \ \wedge\ m \in \mathcal{M}_1(\mathcal{M}) \rhd t' \rhd ch\} \\
& \quad \wedge (\mathcal{M} \lhd v - 1) \in [\![\mathcal{M}]\!]_u^{v-1} \\
& \quad \wedge (\mathcal{M} \lhd t) \in [\![\mathcal{M}]\!]_v \rangle\}
\end{aligned}
$$

Here we use the set $[\![\mathcal{M}]\!]_u^v \subseteq (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty$ for any $\mathcal{M} \in \langle \text{MSC} \rangle$, which is similar to $[\![\mathcal{M}]\!]_u$ except that each element of $[\![\mathcal{M}]\!]_u^v$ constrains the system behavior until time $v \in \mathbb{N}_\infty$:

$$
\begin{aligned}
[\![\mathcal{M}]\!]_u^v \overset{\text{def}}{=} \{ & (\mathcal{M} \lhd v) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\
& \langle \exists (\mathcal{M} \lhd t) : (\mathcal{M} \lhd t) \in [\![\mathcal{M}]\!]_u : \mathcal{M}|_{[u \lhd v]} = \mathcal{M}|_{[u \lhd v]} \ \wedge\ v \leq t \rangle\}
\end{aligned}
$$

Preemptive Loop. The definition of MSC preemption above does not capture the restarting of an interaction in case of the occurrence of a certain message. To handle this case we define the notion of preemptive loop. The intuitive semantics of the preemptive loop of $\mathcal{M}$ for a given message $ch!m$ is that whenever $ch!m$ occurs $\mathcal{M}$ gets interrupted and then the interaction sequence proceeds as specified by $\mathcal{M}$, again with the possibility for preemption. More precisely, we define $[\![\mathcal{M} \Uparrow_{ch!m}]\!]_u$ to equal the greatest fixpoint (with respect to set inclusion) of the following equation:

$$
[\![\mathcal{M} \Uparrow_{ch!m}]\!]_u = [\![\mathcal{M} \overset{ch!m}{\rightarrow} (\mathcal{M} \Uparrow_{ch!m})]\!]_u
$$

The fixpoint exists due to the monotonicity of its defining equation (with respect to set inclusion).

Trigger Composition. By means of the trigger composition operator we can express a temporal relationship between two MSCs $\square$ and $\square$; whenever an interaction sequence corresponding to $\square$ has occurred in the system we specify, then the occurrence of an interaction sequence corresponding to $\square$ is inevitable:

$$\begin{aligned}[\![\square \mapsto \square]\!]_u \stackrel{\text{def}}{=} \{(\square \lhd t) \in (\tilde{C} \times S)^\infty \times \mathbb{N}_\infty : \\ \langle \forall t' \lhd t'' : \infty > t'' \geq t' \geq u : \\ (\square \lhd t'') \in [\![\square]\!]_{t'} \Rightarrow \langle \exists t''' : \infty > t''' > t'' : (\square \lhd t) \in [\![\square]\!]_{t'''} \rangle \rangle \}\end{aligned}$$

Loops. The semantics of a guarded loop, i.e. a loop of the form $\square \uparrow_p$, where p represents a guarding predicate, is the greatest fixpoint (with respect to set inclusion) of the following equation:

$$[\![\square\uparrow_p]\!]_u = [\![(p : (\square \ ; \ \square\uparrow_p)) \mid ((\neg p) : \mathbf{empty})]\!]_u$$

The fixpoint exists because of the monotonicity of its defining equation (with respect to set inclusion); see [9] for the rationale, as well as for other forms of loops (such as bounded finite repetition). On the basis of guarded repetition we can easily define the semantics of $\square$'s infinite repetition (written $\square\uparrow_\infty$) as follows:

$$[\![\square\uparrow_\infty]\!]_u \stackrel{\text{def}}{=} [\![\square\uparrow_{\text{true}}]\!]_u$$

For unbounded repetition we define $[\![\square\uparrow_*]\!]_u \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} [\![\square^i]\!]_u$.

4 Related Work

Suggestions in the literature for MSC dialects abound; [9] contains an extensive list of references. [7,8] defines the standard syntax and semantics for MSC-96. The 2.0 version of the UML[14] has a new interaction model based on MSCs; previous versions adopted a much less powerful notation. LSCs [5] distinguish several interpretations for MSCs (similar to our discussion in [9,10,11]), which allow, in particular, the definition of liveness properties and "anti-scenarios".

None of the above, however, provides operators for treating overlapping scenarios explicitly. By providing the semantic foundation for overlapping using the join operator, we have taken a step towards the independent description of collaborations defining services. Furthermore, although LSCs support complex liveness specifications, we believe that the notion of trigger composition we have borrowed from UNITY's "leadsto" operator (cf. [4]), provides a more accessible way of capturing abstract progress properties. In particular, the combination of trigger composition and join enables separation of concerns in MSC specifications – in the sense of aspect-oriented specification and programming. We refer the reader to [9] for a detailed treatment of further advantages of our model, such as support for MSC refinement and synthesis of component implementations.

5 Conclusions and Outlook

The advent of web services and service-oriented system design in general [12] puts an increasing emphasis on component interaction as the central development aspect. MSCs and similar notations provide a widely adopted means for capturing such interaction patterns in the form of scenarios. MSCs, however, fall short regarding support for important modeling aspects, including overlapping interaction patterns, progress/liveness specifications, and preemption specifications.

In this text, using the ABRACADABRA protocol as an example, we have illustrated the need for corresponding extensions to the MSC notation. In the context of service-oriented system modeling, for instance, specification tools for overlapping interaction patterns are indispensable; the overall system emerges typically as the composition of multiple services such that some components participate in multiple services simultaneously – the corresponding interaction patterns overlap. Similarly, we have shown how to integrate dedicated support for preemption into MSCs as a collaboration-oriented specification notation. Last, but not least, we have also integrated progress specifications into the MSC notation, allowing the developer to express abstract properties such as "whenever one interaction pattern has occurred in the system under consideration, then another interaction pattern is inevitable".

We have also introduced a comprehensive, yet concise mathematical framework for defining the semantics of the extensions we have described. Because our formal model is based on the work in [9], we have at our disposal powerful refinement and synthesis techniques for MSCs even including the extensions described in this text.

Areas for further work include the extension of the treatment of overlapping interaction patterns in the direction of aspect-oriented specifications. Furthermore, the composition operators introduced here need to be substantiated by corresponding tool support; implementation of a corresponding tool prototype is underway.

Acknowledgments. The author is grateful for the reviewers' insightful comments and suggestions. This work was supported by the California Institute for Telecommunications and Information Technology (CAL-(IT)^2).

References

1. Manfred Broy. Some algebraic and functional hocuspocus with ABRACADABRA. Technical Report MIP-8717, Fakultät für Mathematik und Informatik, Universität Passau, 1987. also in: Information and Software Technology 32, 1990, pp. 686–696.
2. Manfred Broy and Ingolf Krüger. Interaction Interfaces – Towards a scientific foundation of a methodological usage of Message Sequence Charts. In J. Staples, M. G. Hinchey, and Shaoying Liu, editors, *Formal Engineering Methods (ICFEM'98)*, pages 2–15. IEEE Computer Society, 1998.

3. Manfred Broy and Ketil Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces and Refinement.* Springer New York, 2001. ISBN 0-387-95073-7.
4. K. Mani Chandy and Jayadev Misra. *Parallel Program Design. A Foundation.* Addison Wesley, 1988.
5. Werner Damm and David Harel. LSCs: Breathing Life into Message Sequence Charts. In *FMOODS'99 IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, 1999.
6. Jan Ellsberger, Dieter Hogrefe, and Amardeo Sarma. *SDL. Formal Object-oriented Language for Communicating Systems.* Prentice Hall, 1998.
7. ITU-TS. Recommendation Z.120 : Message Sequence Chart (MSC). Geneva, 1996.
8. ITU-TS. Recommendation Z.120 : Annex B. Geneva, 1998.
9. Ingolf Krüger. *Distributed System Design with Message Sequence Charts.* PhD thesis, Technische Universität München, 2000.
10. Ingolf Krüger. Notational and Methodical Issues in Forward Engineering with MSCs. In Tarja Systä, editor, *Proceedings of OOPSLA 2000 Workshop: Scenario-based round trip engineering.* Tampere University of Technology, Software Systems Laboratory, Report 20, 2000.
11. Ingolf Krüger, Radu Grosu, Peter Scholz, and Manfred Broy. From MSCs to statecharts. In Franz J. Rammig, editor, *Distributed and Parallel Embedded Systems*, pages 61–71. Kluwer Academic Publishers, 1999.
12. Ingolf H. Krüger. Specifying Services with UML and UML-RT. Foundations, Challenges and Limitations. *Electronic Notes in Theoretical Computer Science*, 65(7), 2002.
13. Leslie Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
14. U2 Partners. Revised submission to OMG RFPs ad/00-09-01 and ad/00-09-02: Unified Modeling Language 2.0 Proposal. Version 0.671 (draft). available at `http://www.u2-partners.org/artifacts.htm`, 2002.
15. Michel Adriaan Reniers. *Message Sequence Chart. Syntax and Semantics.* PhD thesis, Eindhoven University of Technology, 1999.
16. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, Massachusetts, USA, 1999.
17. Robert Stephens. A Survey of Stream Processing. *Acta Informatica*, 34:491–541, 1997.

Index