Lecture Notes in Computer Science 1912

Yuri Gurevich Philipp W. Kutter
Martin Odersky Lothar Thiele (Eds.)

Abstract State Machines

Theory and Applications

International Workshop, ASM 2000
Monte Verità, Switzerland, March 2000
Proceedings

Springer

Lecture Notes in Computer Science 1912
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

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

Yuri Gurevich Philipp W. Kutter
Martin Odersky Lothar Thiele (Eds.)

Abstract State Machines

Theory and Applications

International Workshop, ASM 2000
Monte Verità, Switzerland, March 19-24, 2000
Proceedings

Springer

Series Editors

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

Volume Editors

Yuri Gurevich
Microsoft Research
One Microsoft Way, Redmond, WA 98052, USA
E-mail: gurevich@microsoft.com

Philipp W. Kutter
ETH Zürich, Institute TIK
Gloriastr. 35, 8092 Zürich, Switzerland
E-mail: philipp@kutter.org

Martin Odersky
EPFL Lausanne, DI
IN-Ecublens, 1015 Lausanne, Switzerland
E-mail: martin.odersky@epfl.ch

Lothar Thiele
ETH Zürich, Institute TIK
Gloriastr. 35, 8092 Zürich, Switzerland
E-mail: thiele@tik.ee.ethz.ch

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Abstract state machines : theory and applications ; international workshop ; proceedings / ASM 2000, Monte Verità, Switzerland, March 19 - 24, 2000. Yuri Gurevich . . . (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ; Singapore ; Tokyo : Springer, 2000
(Lecture notes in computer science ; Vol. 1912)
ISBN 3-540-67959-6

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

ISSN 0302-9743
ISBN 3-540-67959-6 Springer-Verlag Berlin Heidelberg New York

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

Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna
Printed on acid-free paper SPIN: 10722735 06/3142 5 4 3 2 1 0

Preface

The ASM 2000 workshop was held in the conference center of the Swiss Federal Institute of Technology (ETH) at Monte Verità, Canton Ticino, March 19-24, 2000.

The ASM formalism was proposed together with the thesis that it is suitable to model arbitrary computer systems on arbitrary abstraction levels. ASMs have been successfully used to analyze and specify various hardware and software systems including numerous computer languages.

The aim of the workshop was to bring together domain-experts, using ASMs as a practical specification method, and theorists working with ASMs and related methods. In addition the workshop served as a forum on theoretical and practical topics that relate to ASMs in a broad sense. Three tutorials including hands-on experience with tools were organized by U. Glässer and G. del Castillo (on the topic "Specifying Concurrent Systems with ASMs"), H. Rüss and N. Shankar (on the topic "A Tutorial Introduction to PVS"), M. Anlauff, P.W. Kutter, and A. Pierantonio (on the topic "Developing Domain Specific Languages").

In response to the organization committee's call for papers, 30 papers were submitted, each of which was independently reviewed by four members of the program committee. This volume presents a selection of 12 of the refereed papers and two reports on industrial ASM application at Siemens AG and Microsoft Research, together with contributions based on the invited talks given by A. Blass (University of Michigan), E. Börger (University of Pisa), G. Goos (University of Karlsruhe), M. Odersky (Swiss Federal Institute of Technology (EPFL), Lausanne), W. Reisig (Humboldt University Berlin), and N. Shankar (SRI International). The introduction written by E. Börger gives an overview on ASM research from the beginning to the present.

On behalf of the program committee, we would like to express our appreciation to the six lecturers who accepted our invitation to speak, to the tutorial organizers, to all the authors who submitted papers to ASM 2000 and to the Centro Stefano Franscini (CSF) at Monte Verità and the local organizers who made the workshop possible.

June 2000 Y.Gurevich, P.W.Kutter, M.Odersky, and L. Thiele

Organization

ASM 2000 took place from March 19th to 24th at the conference center Monte Verità of the Swiss Federal Institute of Technology (ETH).

Organization Committee

Yuri Gurevich	(Microsoft Research, Redmond, WA, USA)
Philipp W. Kutter	(ETH, Zürich, Switzerland)
Martin Odersky	(EPFL, Lausanne, Switzerland)
Lothar Thiele	(ETH, Zürich, Switzerland)

Program Committee

Andreas Blass	(University of Michigan, USA)
Egon Börger	(University of Pisa, Italy)
Uwe Glässer	(HNI Paderborn, Germany)
Carla P. Gomes	(Cornel University/Rome Labs, USA)
Georg Gottlob	(TU Wien, Austria)
Erich Grädel	(University of Aachen, Germany)
Irene Guessarian	(LIAFA/University Paris6, France)
Yuri Gurevich	(co-chair, Microsoft Research, USA)
Jim Huggins	(Kettering University, USA)
Stefan Jähnichen	(GMD FIRST Berlin, Germany)
Hans Langmaack	(University of Kiehl, Germany)
Larry Moss	(Indiana University, USA)
Peter Mosses	(BRICS, Denmark)
Martin Odersky	(co-chair, EPFL, Switzerland)
Alfonso Pierantonio	(University of L'Aquila, Italy)
Arnd Poetzsch-Heffter	(FernUni Hagen, Germany)
Elvinia Riccobene	(University of Catania, Italy)
Dean Rosenzweig	(University of Zagreb, Croatia)
Harald Ruess	(SRI International, USA)
Daniel Schweizer	(UBS Zürich, Switzerland)
Anatol Slissenko	(University Paris 12, France)
Lothar Thiele	(co-chair, ETH Zürich, Switzerland)
Richard Waldinger	(SRI International, USA)
Alexandre V. Zamulin	(Russian Academy of Science, Russia)
Wolf Zimmermann	(University of Karlsruhe, Germany)

Local Organizers

Bernhard Bron	(ETH Zürich, Switzerland)
Samarjit Chakraborty	(ETH Zürich, Switzerland)
Christoph Denzler	(ETH Zürich, Switzerland)
Felix Etter	(ETH Zürich, Switzerland)
Monica Fricker	(ETH Zürich, Switzerland)
Philipp W. Kutter	(ETH Zürich, Switzerland)
Milan Tadjan	(ETH Zürich, Switzerland)

Sponsoring Institutions

Swiss Federal Institute of Technology, Zürich, Switzerland

Microsoft Research, Redmond, WA, USA

Swiss National Science Foundation (SNF)

Blue Capital Inc., Zürich, Switzerland

Table of Contents

Automatic Verification and Model Checking

Industrial Applications

Abstract State Machines at the Cusp of the Millenium

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it (Visiting Microsoft Research, Redmond)

Abstract. The ASM'2000 Workshop marks for the ASM method the transition from its adolescence to the maturation period. The goals which have been achieved open new frontiers and put us into the position to embark on new challenges.

1 The Start at the End of the Twentieth Century

We went a long way since the Spring of 1987 when Yuri Gurevich visited Pisa and, in a series of lectures on the fundamental problem of semantics of programming languages, presented the world première of the concept of ASMs (then called dynamic/evolving structures/algebras). He gave the main motivation: reconsider Turing's thesis in the light of the problem of semantics of programs. He illustrated his ideas with examples, in particular specifications of Turing machines, stack machines and some Pascal programs. He gave also proofs of simple properties of these programs. This material appeared a year later in [22]. It was preceded by the first appearance of the ASM Thesis, in embryo in a 1984 technical report [20], and fully spelled out in a notice presented on May 13 of 1985 to the American Mathematical Society [21]. It was accompanied by the first real-world application, namely the dynamic semantics of MODULA-2 [26], and shortly afterwards followed by the ASM treatment of concurrency used to define the semantics of OCCAM [27], which was presented by Gurevich in another series of lectures in Pisa in May 1990. Since then the concept of Abstract State Machines essentially remained stable [23,24][1]and triggered hundreds of publications in various domains including finite model theory, complexity theory and numerous areas of applied computer science, in particular programming languages, database query languages, protocols, architectures and embedded control software [1].

The first attempts to put the bold ASM thesis to the test were focussed on the problem of the dynamics of programming languages known to us, and we came from a purely theoretical background and had no practical, let alone

[1] The initially present construct to shrink domains, which was motivated by concerns about resource bounds, was abandoned because it belongs to garbage collection rather than to high-level specification. Some technical variation was later introduced concerning the treatment of non determinism and of inconsistent update sets.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 1–8, 2000.

industrial, experience. What came out of that is a practical method which exploited ASMs for the development of a full-fledged refinement approach which enabled us to rigorously define and analyse the dynamic semantics of real-life programming languages and their implementation on virtual or real machines. By now, the covered programming paradigms include the paradigms of all the major modern programming languages. The method of constructing *ground models*, described in [5] where the name primary model (rather than ground model) was used, proved to be mature and was chosen for standardization purposes by the International Standards Organization, see [8,9,11,12,3], and by the International Telecommunication Union, as reported in [19] and in these Proceedings [18].

At the next step, the method was tried out for the specification and verification of machine architectures and protocols. Eventually this was followed by applications to software engineering. Here one starts by constructing a *ground model* for a proposed (or, in the case of reverse engineering, for an existing) software system. Through the entire design process, one refines and/or coarsens the models linking the high level models in a traceable and inspectable way to executable code; see the survey in [6].

The key for the surprisingly fast success of the method lies in (a) the two constituents of the notion of ASM, namely being *abstract* (Abstract State) and *operational* (Abstract Machine) (see the section *Abstract Machines + Abstract State = ASM* in [6] for the historical reconstruction of the confluence of these two concepts[2]) and (b) in the systematic way it offers for practical software development to *separate different concerns*.

The **abstract character** of ASMs allows one, on one side, to tailor the models to the needs or purposes of the design, and, on the other side, to make their rigorous analysis feasible. The latter is due to the freedom to use those proof methods which are appropriate for the present discourse. In other words, the abstraction mechanism, built into the notion of ASM, permits one to make *real* the old dream of well documented and controllable hierarchical system development. Use ASMs to do the following:

- Make the faithfulness of the models, with respect to the design intentions, checkable by direct inspection (falsifiable in the Popperian sense). This holds in particular in requirements engineering for the faithfulness of the ground model with respect to the informally given requirements. The faithfulness becomes checkable by the application domain expert once an ASM model is there (see [16]).
- Link, by hierarchies of stepwise refinements, the high-level definition in a transparent way to its implementation. Here each refinement step is supposed to reflect design decisions one wants to document for future use, e.g. for maintenance purposes or for changes by extensions and modifications.

[2] Notice that the ASM abstract machine is different from the familiar abstract machines. It has built-in parallelism which allows one to abstract from irrelevant sequentialization.

– Make the (mathematical, possibly machine checked) justification of the correctness of a complex design feasible. ASMs offer the necessary rigorous framework for the analysis of run-time properties, at the appropriate level of abstraction, which allows one to prove that the implementation conforms to the high-level specification.

The **operational character** of ASMs provides a sufficient basis for turning the definitions into executable models. These can be used for high-level validation of user scenarios prior to coding or of test beds by means of mental or machine driven experiments. This luckily breaks with the still widely held traditional view that specifications should be, or are by definition, non-executable.

The **separation of different concerns** is incorporated into the techniques provided by the ASM method for constructing different system views and linking them into hierarchies of system levels. Here are the major software development concerns we systematically separate and recombine, making *divide and conquer* the main methodical principle, and it led us very far.

– The separation of orthogonal design decisions is the most important one in practice. It is made possible by the most general abstraction and refinement capabilities of ASMs, and it is motivated by the necessity to keep the design space open as long as possible and to structure it, for "design for change" and for modular development.
– The separation of design from analysis corrects the long standing tradition of identifying "rigorous" with "formalized in logic". This tradition is one of the reasons why the so called formal methods have not really had a strong impact on practical software development.
– The separation, within the analysis, of experimental validation from mathematical verification is possible with ASMs because once the ASM models have been made executable, they can be used for simulation. The simulation of a higher-level model can be performed prior to writing the final code. It can also enhance testing the code, so that the correspondence of the implementation to the abstract specification can be checked.
– The separation of different degrees of detail within verification allows one to adapt the justification of a design to the current development stage. ASMs give the means to differentiate justifying a design to domain experts from justification in terms of mechanical reasoning systems. Further, one should distinguish between interactive logical systems and fully automated tools like model checkers or automatic theorem provers.
– The separation of mathematical concerns within verification. For example, one may want to split the proof of a property P for a complex system S into three steps, namely:
  - prove P for an abstract model M of S under an appropriate assumption A,
  - refine M to S so that S implements M correctly,
  - prove that S satisfies the assumption A.

Experience shows that it is not only easier to prove properties for complex systems in this way, but this splitting of proof obligations often is the only known way to show that a run-time system works the way it is supposed to. A characteristic example are the ASM-based proofs of the correctness of compilation schemes [13,10,17] and the implementation of those schemes by provably correct real-life compilers, a problem which is addressed in these Proceedings. By the way, collecting such justificational evidence yields an interesting byproduct: a detailed analysis of the design itself.

Most of these lines of research and principles were present, in embryo, already at the first international ASM workshop held as early as 1994, as part of the IFIP World Computer Congress in Hamburg, Germany [30]. Although the applications which appeared there were limited by our academic experience and were largely motivated by an effort to critically test the ASM thesis, it is not incidental that right from the beginning, by making good use of the freedom of abstraction offered by the ASM concept, we were naturally led to "separate and combine" design levels, design and analysis, verification and validation, and degrees of detail in verification. Concerning validation, let us note that the first tools for making ASMs executable date back to 1990; see the Prolog-based compiler for ASMs Angelika Kappel developed in [29], to execute my ASMs for PROLOG [8], and the Michigan interpreter mentioned in [23][3].

Only 5 years later, at the ASM workshop which was held as part of the International Formal Methods Conference FM'99 in Toulouse, France[4], one can observe (see again the survey [6] for details) that

- the theory of ASMs is richly developed,
- the applications include industrially successful standardization and software engineering projects, some of which become publicly visible for the first time in these Proceedings [15,4],
- there is a proliferation of different tools[5], most of them developed in academia, for both the experimental validation and the machine supported verification of ASMs, providing execution mechanisms for ASMs (via interpretation or compilation) and links to verification systems like PVS, KIV and model checkers.

[3] Here is what Jim Huggins wrote to me on June 4, 2000, about the details of the history: "Yuri taught a course in programming language design at Michigan during the Fall of 1990. Of course, he introduced ASMs in the class. A very sharp young undergraduate named Ben Harrison wrote a bare-bones interpreter for ASMs in LISP in a weekend and distributed it to the class. Yuri was impressed enough with Harrison that he hired him to write a full-fledged ASM interpreter, this time in C. Ben built the core of the interpreter in May-June 1991. At that point the interpreter was handed over to me, and I worked on it for 3 years or so, finishing up the rest of the unfinished business at that time. In 1994 development was handed over to Raghu Mani, who worked on it for a couple of years...". Let me add that Raghu's task was to upgrade the interpreter for one-thread ASMs to multi-agent ASMs.

[4] In between, ASM workshops had been held in Paderborn (May 1996), Cannes (June 1997, June 1998) and Magdeburg (September 1998).

All of these themes are reflected in the rich program of ASM'2000. These Proceedings, which constitute the first book entirely devoted to ASMs, document what has been achieved in the first decade after the formulation of the ASM concept. We see confirmation of our conviction expressed already at the first ASM workshop in 1994, namely that (paraphrased) "the extraordinary potential of the ASM method will change drastically the industrial future of formal specifications" [5, pg.393].

2 New Frontiers

The experience accumulated with the ASM concept, and with the method which has been developed for its use, did change the way we think about high-level software design and analysis. Now we have to actualize this vision to make it work for established software development disciplines, at a large scale. The achievements of the last decade open new frontiers and put us into the position to face the new challenges.

Through the extensive ASM modeling, validation and verification work of the past decade, the ASM thesis was experimentally confirmed. But this year brought us a theoretical explanation of the observed phenomenon, namely via a proof [25] that the sequential version of the thesis follows from three fundamental system theory axioms. Once established, the thesis allows one to draw conclusions of practical importance, as is illustrated by an example in these Proceedings [7]: the thesis guarantees that there is no loss of generality in substituting the fundamental but vague UML concepts of action and activity by the mathematically rigorous concepts of ASM step and ASM run. It seems that in UML [31,2] the meanings of action/activity were intentionally left unspecified, namely to leave the space of possible implementations as open as possible. But this was achieved at the price of making it difficult to control the implications the concepts have in the context of the event-driven *run-to-completion* scheme, in particular concerning the possibly numerous and nested exit/entry actions, coming through interrupts, and concerning the launch and abortion of internal activities.

On the practical side we have to take advantage of the experience, acquired with building tools for executing ASMs, to develop an entire tool *environment* which is also industrially satisfactory. It has to support the different activities of defining, transforming (by refinements and by code generation) and analysing ASM models (by testing, via visualization supported simulation, and by verification). The tool environment has to enable us to capture the design knowledge in a rigorous, electronically available and reusable way, and to achieve this goal it must be integrated into established design flows and their tool environments. The integration potential of ASMs, as a universal model of computation which

[5] When I was working on this introduction, a message from Prof. Igor Soloviev, of St.Petersburg State University, arrived: "One of my students, Andrew Usov, has written an ASM interpreter. It is implemented as an java-applet designed to run under an internet-browser (it has been tested under IE 5.0 and NS 4.7)."

is well established by the accumulated experimental evidence and by the theoretical explanation we have by now for the ASM thesis, is helpful to capture the overall behavior of a complex system by combined use of whatever rigorous descriptions are appropriate and mandatory in established design approaches (static, dynamic, functional, state-based, object-oriented, etc.).

This is a difficult and probably long way to go, "a ridge walk between freedom and discipline, creativity and pattern oriented design, generality and specialization, expressability and limitations by tool support" [6]. But it is worth the effort. Here are some among other challenging problems where I see a large potential for fruitful exploitation of the ASM method.

- If we succeed to construct paradigmatic and parameterized ASM components and to extract (de)composition techniques that can be made available in libraries, the "codeless" form of ASM programming will help porting application programs from one platform or language to another and can lead to fruitful applications for plug-and-play software technology.
- If we succeed to exploit ASMs for defining and implementing methods for generating test suites from high-level specifications, this will turn a dark and at present overwhelming part of software development into an intellectually challenging and methodologically well supported task of enormous practical value. Indeed using ASMs one can solve the crucial and essentially creative part of test case selection, given that this selection is driven typically by appplication domain expert knowledge and thus can be formulated using the ASM ground model. Similarly the ground model supports solving the oracle problem of testing: the expected output, which has to be compared with the execution output, can be defined using the ground model specification (which is independent of the programming language where the system will be encoded)[6].
- If we exploit ASMs to enhance current (mostly signature oriented) software architecture description techniques by adding to the structural definitions also relevant semantical content, we will solve a widely felt need for building reliably reconfigurable *conceptual* and *module interconnection architectures* [28].
- If we succeed to exploit the atomic transaction nature of the notion of ASM-step to model practically useful patterns for communication and synchronization of multi-agent ASMs, typically based on shared memory or on message passing, then we will contribute to solve a crucial problem of distributed computing.

On the theoretical side a model and a proof theory of ASMs are needed. We need definitions which capture and enhance the practical refinement schemes we have used with success for ASMs, together with useful proof principles which can be built into state-of-the-art mechanical verification systems (for some steps in this direction see the contributions to PVS and model checking in these Proceedings and [32,14]). The proof theory we need should alleviate the verification effort encountered in practical applications, namely by offering structuring and layering of proof obligations which avoid the bottleneck of a priori fixed levels

[6] A similar remark applies also to static testing (code inspection) where one has to formulate the properties to be checked.

of overwhelming proof details. We need to find the right way to exploit the notion of monitored (real-valued) function for connecting the discrete ASM world to the continuous world of control theory. We need to help in building models for mobile computing. We badly need to extract the inherent object oriented features of ASMs, which are visible in the concept of ASM agents and of their state, to make them explicitly and syntactically available, adapted to established object-oriented programming techniques.

These Proceedings contain numerous contributions where the mentioned issues are raised and thus constitute a good point of departure to help solving the challenging problems which are waiting for us.

References

1. Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.
2. *Rational Software Corporation, Unified Modeling Language UML, version 1.3*, 1999.
3. ISO/IEC 13211-1. Prolog-Part 1: General Core. In *Information Technology-Programming Languages*. International Standards Organization, 1995.
4. M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using ASMs at Microsoft: A Case Study. In *This volume*.
5. E. Börger. Logic Programming: The Evolving Algebra Approach. In B. Pehrson and I. Simon, editor, *IFIP 13th World Computer Congress*, number I (Technology/Foundations), pages 391–395. Elsevier, 1994.
6. E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
7. E. Börger, A. Cavarra, and E. Riccobene. Modeling the Dynamics of UML State Machines. In *This volume*.
8. E. Börger and K. Dässler. PROLOG. DIN Papers for Discussion. Report 58, ISO/IEC JTC1 SC22 WG17, April 1990.
9. E. Börger and B. Demoen. The view on database updates in Standard Prolog: a proposal and a rationale. Report 74, ISO/IEC JTC1 SC22 WG17 , February 1991.
10. E. Börger and I. Durdanovic. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, (39(1)):52–92, 1996.
11. E. Börger and D. Rosenzweig. An Analysis of Prolog Database Views and Their Uniform Implementation. In K. Dässler and R. Scowen, editor, *Prolog. Paris Papers–2*, number 80, pages 87–130. National Physical Laboratory, Middlesex, July 1991.
12. E. Börger and D. Rosenzweig. The Mathematics of Set Predicates in Prolog. In K. Dässler and R. Scowen, editor, *Prolog. Copenhagen Papers–2*, number 105, pages 33–42. National Physical Laboratory, Middlesex, 1993.
13. E. Börger and D. Rosenzweig. The WAM–Definition and Compiler Correctness. In Ch. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. Elsevier Science B.V./North–Holland, 1995.
14. E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editor, *Gurevich Festschrift CSL 2000*, LNCS. Springer-Verlag, 2000. (In print).
15. E. Börger, J. Schmid, and P. Päppinghaus. Report on a Practical Application of ASMs in Software Design. In *This volume*.

16. E. Börger, J. Schmid, and E. Riccobene. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *J. Universal Computer Science*, 2000. Special Requirement Engineering Issue, to appear.
17. E. Börger, J. Schmid, R. Stärk, and W. Schulte. *Java and the Java Virtual Machine*. Springer-Verlag, 2000. to appear.
18. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: a Compilation Approach Based on an Abstract SDL Machine. In *This volume*.
19. U. Glässer, R. Gotzhein, and A. Prinz. Towards a New Formal SDL Semantics Based on Abstract State Machines. In R. Dssouli and G.v. Bochmann and Y.Lahav , editor, *SDL'99 - The Next Millenium (Proc. of the 9th SDL FORUM)*. Elsevier Science B.V., 1999.
20. Y. Gurevich. Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs. Technical Report CRL-TR-36-84, University of Michigan, Computing Research Lab, 1984.
21. Y. Gurevich. A New Thesis. *Notices of the American Mathematical Society*, page 317, 1985. abstract 85T-68-203, received May 13.
22. Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.
23. Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 225–234. World Scientific, 1993. A reprint of the article in the Bulletin of the European Association for Theoretical Computer Science, Number 35 (1991), pp.71-82.
24. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
25. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1, 2000.
26. Y. Gurevich and J. Morris. Algebraic operational semantics and modula-2. In Börger, E. et al., editor, *CSL'87, 1st Workshop on Computer Science Logic*, number 329 in LNCS, pages 81–101. Springer-Verlag, 1988.
27. Y. Gurevich and L. A. Moss. Algebraic operational semantics and occam. In Börger, E. et al., editor, *CSL'89, 3d Workshop on Computer Science Logic*, number 440 in LNCS, pages 176–192. Springer-Verlag, 1990.
28. C. Hofmeister, R.L. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.
29. A.M. Kappel. Implementation of Dynamic Algebras with an Application to Prolog. Master's thesis, CS Dept., University of Dortmund, Germany, November 1990. An extended abstract "Executable Specifications based on Dynamic Algebras" appeared in A. Voronkov (ed.): Logic Programming and Automated Reasoning, volume 698 of LNAI, Springer, 1993, pages 229-240.
30. B. Pehrson and I. Simon. *IFIP 13th World Computer Congress. Vol.I: Technology/Foundations*. Elsevier, 1994.
31. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
32. G. Schellhorn. Verifikation abstrakter Zustandsmaschinen. Phd thesis, CS Dept., University of Ulm, Germany, 1999. For an English version consult www.informatik.uni-ulm.de/pm/kiv/papers/verif-asms-english.ps.gz.

Abstract State Machines and Pure Mathematics

Andreas Blass*

University of Michigan, USA
ablass@umich.edu

Abstract. We discuss connections, similarities, and differences between the concepts and issues arising in the study of abstract state machines and those arising in pure mathematics, particularly in set theory and logic. Among the topics from pure mathematics are the foundational role of set theory, permutation models of set theory without the axiom of choice, and interpretations (between theories or vocabularies) regarded as transformations acting on structures.

1 Introduction

The purpose of this paper is to describe some connections between the theory of abstract state machines (ASM's) and concepts from pure mathematics. These connections are of three general sorts.

First, there are direct uses of mathematical concepts in ASM's. A well known instance of this is the use of structures, in the sense of first-order logic, in the very definition of ASM's. A less well known instance is the use of interpretations, also in the sense of first-order logic, to describe transitions of ASM's as well as certain sorts of simulations.

Second, there are modifications of mathematical concepts, adapting them to the purposes of computation theory and ASM's. As an example of this, we discuss the ASM analog of the set-theoretic concept of permutation model.

Finally, there are analogies between ASM's and some aspects of pure mathematics. In this connection, we discuss the multifaceted philosophical issue of "universality": Do ASM's provide a universal framework for descriptions of algorithms in the same sense that set theory provides a universal framework for mathematical proofs? In this connection, we also discuss the value of explicit formalization and the role of definitions. We comment briefly on possible alternative "foundations" based on, for example, ordered lists or multisets instead of sets.

We also discuss the issue of "objects whose identity doesn't matter," which arises in connection with the `import` rules of ASM's and also in various contexts in pure mathematics.

* Preparation of this paper was partially supported by a grant from Microsoft Corporation. The opinions expressed here are, however, entirely my own.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 9–21, 2000.

Acknowledgements. I thank the organizers of the Monte Verità conference for inviting me to present the talk on which this paper is based. I also thank Yuri Gurevich for his suggestions for improving this paper and for many helpful and informative discussions about ASM's and other aspects of computer science.

2 Structures and Interpretations

From the very beginning, the concept of abstract state machines has been closely linked to pure mathematics. Indeed, one of the central ideas is to model states of computations as structures in the sense of mathematical logic. Another central idea is that an ASM program describes one step in a computation, not the entire iteration. Because of this, the programs themselves correspond to a familiar concept from first-order logic, that of interpretation.

Interpretations of one vocabulary or theory in another were introduced in [17] for the purpose of deducing undecidability of many theories from essential undecidability of one theory. See also [15, Section 4.7], whose formulation we follow here. In general, an interpretation of a vocabulary Υ in a theory T (with vocabulary Υ_T) consists of

- a unary predicate U in Υ_T,
- for each function symbol F of Υ a function symbol F_I of Υ_T, and
- for each predicate symbol P of Υ a predicate symbol P_I of Υ_T (equality does not count as a predicate symbol here but as a logical symbol),

such that it is provable in T that U is nonempty and closed under all the F_I's. Thus, given a model M for T, we obtain a structure for Υ by taking as its base set the extent in M of U and as its functions and predicates the (restrictions of the) interpretations in M of the corresponding symbols of Υ_T. The concept of interpretation is often extended by saying "interpretation in T" when one really means "interpretation in an extension by definitions of T." Then U, F_I, and P_I need not be primitive symbols of the vocabulary Υ_T but could be given by formulas in that vocabulary. (For historical accuracy, I should mention that the definition of "interpretation" in [17] already included extensions by definitions but did not include the unary predicate U; the latter was treated separately under the name of "relativization.")

Consider the special case of an interpretation where the theory T has no axioms (so the interpretation can be applied to arbitrary Υ_T-structures), where $\Upsilon = \Upsilon_T$, and where U is identically true (so applying the interpretation doesn't change the vocabulary or the base set of a structure). The transformations of Υ-structures obtainable by such transformations are just those obtainable by executing one step of an ASM with vocabulary Υ, provided the ASM doesn't create new elements and is non-distributed. (In the distributed case, the interpretations should be restricted so that the steps they describe can each be executed by a single agent.)

From this point of view, one could say that an ASM is "just a special sort of interpretation," but the word "just" here is unjust. Though formally the

same, ASM's are intended to be used quite differently from interpretations: An ASM is to be applied iteratively (for a number of steps not usually known in advance). For interpretations in general, there is a notion of composition, but only interpretations from a vocabulary (or theory) to itself can be repeated arbitrarily often.

It may be noteworthy that, in the correspondence between ASM's and interpretations, the sequential ASM's correspond to quantifier-free interpretations. Thus, the same restriction is quite natural from both the computational and the logical points of view.

Can interpretations contribute anything to our understanding of ASM's? In a sense, they already did, at least if one considers interpretations in a generalized sense that has become common in model theory. There (following an idea that I believe was introduced by Shelah), one more or less automatically enlarges structures to include, in addition to their actual elements,

- tuples of elements and
- equivalence classes with respect to definable equivalence relations.

See for example the M^{eq} construction in [13, page 10]. Then interpretations can use these additional elements; thus for example the standard definition of the integers in terms of natural numbers (as equivalence classes of pairs) amounts to an interpretation. Some of the central ideas and results of [2] first appeared, in notes of Shelah, in the context of interpretations of just this sort. (They weren't explicitly called interpretations, but the influence of M^{eq} was visible.) Later, this material was clarified by expressing it in terms of ASM's. The distinctive set-theoretic context of [2] was motivated, at least in part, by the desire for a natural framework in which tuples and equivalence classes are easily handled.

Might interpretations contribute something more to ASM theory? What about interpretations between different vocabularies, which are therefore not iterable? Perhaps these will provide a useful point of view for such operations as setting up an initial state for an ASM. Often the input to a computation is only part of what should be present in the initial state of a computation. Adding the extra material to convert the input into the initial state is, in some cases, an interpretation in the generalized (M^{eq}) sense discussed above. In other cases, this preparation is more complicated, for example adding a whole set-theoretic superstructure as in [2]. It is not clear whether the notion of interpretation can be usefully stretched to cover such cases as well.

In addition to setting up the initial state for a computation, interpretations might be appropriate for extracting the "answer" from the final state of a computation. More generally, one can imagine many sorts of interfaces — not only input/output but more general interfaces between computations or between parts of a computation — as described by interpretations.

It also seems that interpretations may provide a good way to describe the connection between two ASM's that "compute the same thing in different ways." The idea here is that certain states, the "pose" states, in a run of the one machine are obtainable from those of the other machine by a uniform interpretation. (Between successive pose states, each machine may go through states that have

no strict counterpart in the other machine, states that depend on the details of how the machine does its work.) The interpretation tells how what one machine does is to be seen from the perspective of the other.

Finally, since interpretations are usually (in mathematical logic) regarded primarily as syntactic transformations rather than semantical ones, I should add a few words about the syntactic aspect. An interpretation of a vocabulary Υ in a theory T provides a translation of terms and formulas from Υ to the vocabulary of T. The translation consists of replacing each function symbol F or predicate symbol P in the given term or formula with F_I or P_I and restricting all quantifiers to U_I. Then the translation φ^I of a sentence φ is true in a model M of T if and only if the sentence φ itself is true in the Υ-structure M_I obtained from M via the interpretation.

Thinking of interpretations primarily on the semantic level, transforming models M of T into Υ-structures M_I, we can describe the syntactic transformation $\varphi \mapsto \varphi^I$ as producing weakest preconditions. That is, φ^I expresses, in a structure M, exactly what is needed in order that M_I satisfy φ. Here is a toy example to indicate what goes on in general.

Consider the update rule $c := d$. The corresponding interpretation has $P_I = P$ and $F_I = F$ for all predicate and function symbols other than c, and $c_I = d$ (and U_I is the identically true predicate). If M is a structure, then M_I for this interpretation is the same as M except that it gives c the value that d had in M, i.e., it is the sequel of M with respect to the rule $c := d$. Now consider a formula φ, the simplest relevant example being $P(c)$ for a unary predicate P. Its translation φ^I under our interpretation is obtained simply by replacing every occurrence of c by d, so in our simple example $P(c)^I$ is $P(d)$. And this φ^I is exactly what M must satisfy in order to guarantee that M_I satisfies φ.

3 Permutation Models

In this section, I'll describe a technique for describing and analyzing the idea of "not permitting arbitrary choices," both in set theory and in computation theory.

In 1922, Fraenkel [5] constructed models of set theory (with infinitely many urelements, also called atoms) in which the axiom of choice is false. The central idea is that all the atoms in a model of set theory "look alike"; more precisely, any permutation of the atoms induces an automorphism of the whole set-theoretic universe. Fraenkel formed the subuniverse of "sufficiently symmetric" sets, and showed that it satisfies all the usual axioms of set theory except the axiom of choice. ("Usual" is an anachronism here, since Fraenkel himself had just recently introduced the replacement axiom and given a precise formulation of Zermelo's separation axiom. But the axiom system, known as Zermelo-Fraenkel set theory, ZF, is certainly the usual one nowadays.) More precisely, Fraenkel's model consisted of those sets x which depend on only a finite subset F of the atoms, in the sense that any permutation of the atoms fixing those in F will, when extended to an automorphism of the whole universe, also fix x. (Subsequently, other authors

studied other notions of "sufficiently symmetric" in order to obtain independence results concerning weak forms of the axiom of choice, but these other notions will not be relevant here.)

A modification of Fraenkel's idea is at the heart of the main results of [2]. This paper introduced a rather liberal model of choiceless polynomial time computation. The model is an ASM whose initial state is obtained by building a universe of (hereditarily) finite sets over the input structure regarded as consisting of atoms; this makes many sorts of data structures available for the computation. In addition, parallelism is allowed both in `do-for-all` rules and in some set-formation constructions. It is shown, however, that such apparently simple things as the parity of the number of elements in an unstructured set cannot be computed in polynomial time, even in this liberal model, as long as arbitrary choices are prohibited. The proof of this theorem uses the hypothesis of choicelessness by deducing that, if the computation "uses" a set x, then it must also use all sets $\pi(x)$ obtainable from x by automorphisms of the set-theoretic structure, i.e., by permutations of the atoms of the input structure. The polynomial time hypothesis then ensures that there cannot be too many of these $\pi(x)$'s. The essential combinatorial step in the proof is a lemma allowing us to infer from "not too many $\pi(x)$'s" to x being fixed by all permutations that fix a certain number of atoms. In contrast to Fraenkel's situation, "a certain number" cannot be taken to be simply "finitely many," as there are only finitely many atoms altogether in the present situation. Instead, a quantitative estimate is needed, which depends on the ASM program under consideration. Thus, the "infinite vs. finite" dichotomy exploited by Fraenkel has been replaced by "large finite (growing as the input structure grows) vs. small finite (depending only on the ASM program).

A further refinement of Fraenkel's idea occurs in [14], where Shelah proves a zero-one law for properties of graphs computable in choiceless polynomial time. For the purposes of permutation arguments, there is a crucial difference between the unstructured (or the slightly more general "colored") sets considered in [2] and the graphs considered in [14]: Almost all finite graphs have no non-trivial automorphisms. So at first it would seem that symmetry arguments cannot be applied to computations taking random finite graphs as inputs. Shelah circumvented this problem by working not with automorphisms but with partial automorphisms of graphs. This required a rather delicate development, with careful attention to the sizes of the domains of these partial automorphisms, to ensure that the behavior of a choiceless polynomial time computation is invariant (in a suitable sense) under partial automorphisms of the input. For the (rather complicated) details, we refer the reader to [14] or to an easier to read (we hope) exposition that Yuri Gurevich and I are preparing.

4 ASM's and Set Theory

At a 1993 meeting in Dagstuhl, after hearing several talks about formulating various algorithms as ASM's (then still called "evolving algebras"), I made a comment along the following lines:

> It seems to me that expressing algorithms in the ASM formalism is rather like formalizing mathematical proofs in ZFC. At the beginning, one needs a number of examples of such formalizations, but after a while it becomes clear that any "reasonable" algorithm can be written as an ASM, just as any "reasonable" proof can be formalized in ZFC. And after a while, there is little point in actually carrying out the formalizations just in order to verify formalizability (in either situation) unless and until a genuinely problematic case arises.

In the case of sequential algorithms, it seems fair to say that this comment has been confirmed by subsequent experience. Better yet, it is confirmed by the main result of Gurevich [8], which asserts that any sequential algorithm can be expressed by an ASM provided it satisfies certain very natural postulates.

I'll comment later on the situation with non-sequential algorithms, but first let me point out some differences between the roles of ASM's and of ZFC even in the sequential case.

The most obvious difference is that most of us have never seen a non-trivial proof fully formalized in ZFC (nor would we want to see one), but we have seen non-trivial algorithms fully formalized as ASM's. ASM's are, by design, reasonably close to our mental images of algorithms; ZFC is similarly close to mental images of only small parts of mathematics (primarily set theory, and not even all of that). To write out the proof of, say, the commutative law for real addition in the primitive language of set theory would be a very time-consuming exercise for which I see no value at all.

In contrast, writing out explicit ASM programs for various algorithms is not only feasible but worthwhile. It has, for example, led to the detection of errors or ambiguities in specifications of programming languages and computer systems.

This distinction is reflected in a second difference between ASM and ZFC formalizations: the ubiquity of *definitions* in the development of mathematics on a set-theoretic basis. Such a development begins with axioms written in purely set-theoretic notation, but it soon introduces other concepts by definition, and it is not at all unusual for a concept to be many definitional layers removed from the primitive set-theoretic notions. Think, for example of the real numbers, defined as equivalence classes of Cauchy sequences of rational numbers. The definition of $\mathbb{R}$ sits at the top of a rather high tower (or pile) of other definitions.

Something roughly analogous happens in ASM's, in the idea of successive refinement, working from a high-level description of an algorithm down to a specific implementation. But the analogy is only a rough one for two reasons. First, the tower that connects low-level and high-level versions of an algorithm consists of ASM's at every level. The tower connecting ZFC to, say, the theory of partial differential equations, departs from ZFC formalization (in the strict

sense) as soon as one gets above the bottom level; the higher levels are formalized, if at all, in first-order theories obtained (formally) by adding definitions as new axioms. Second, the refinement process is a serious object of study in the ASM world, whereas in the set-theoretic world the process of definition is usually swept under the rug — so much so that many logicians would be taken aback by my comment a moment ago that differential equation theory is formalized not in ZFC but in an extension by definitions. They would probably say "what's the difference?" and accuse me of splitting hairs. The only authors I know of who have made a serious study of the role of definitions in set theory are Morse [12], who carefully analyzes the permissible syntax (far more general than what is usually considered), and Leśniewski [11] who actually uses definitions in a non-conservative way, as an elegant formulation of existence axioms.

This suggests a question for ASM theory: Can one push the refinement process as far into the background as set-theorists have pushed definitions (and if not then how far can one push it)? That is, can one arrange things so that all one has to write is a high-level specification and the sequence of "definitions" of its functions in terms of lower levels? The idea would be that the corresponding low-level ASM would either be produced automatically or would (as in the set-theoretic situation) be entirely irrelevant. And of course the high-level ASM and the definitions together should be significantly shorter than the low-level ASM they describe.

A third difference between ASM's and ZFC is the use of ASM's for detecting and correcting errors in intuitive specifications, programs, etc. As far as I know, formalization in ZFC has not played a role in detection and correction of errors in mathematical proofs. Errors certainly occur, and they are detected and corrected. (For particularly embarrassing examples, see the footnotes on page 118 of [9] and the introduction of [16]. I emphasize that the authors of [9] and [16] did not commit the errors but conveniently summarized them.) But the detection and correction seems to proceed quite independently of formalization. It is based on intuitive understanding and is therefore not very systematic. There are, of course, projects to formalize and systematically check, by computer, various parts of mathematics. But these seem to be of more interest, at the moment, to computer scientists than to typical mathematicians. The latter seem to be generally quite willing to rely on intuitive understanding rather than formal checking.

A fourth difference between the role of ASM's in formalizing algorithms and the role of ZFC in formalizing proofs is that in the latter context there is (to the best of my knowledge) no analog of [8]. That is, there is no general criterion guaranteeing that any proof, subject to some natural constraints, is formalizable in ZFC. Mathematicians other than set theorists generally take on faith that anything they would consider a correct proof can be formalized in ZFC. Set theorists are aware of possible difficulties (with universe-sized sets), but we are confident that we could recognize any such difficulty and determine whether a proposed argument really goes beyond ZFC (without actually writing out a formalization).

Having promised to comment on the situation for non-sequential algorithms, let me say that the situation there is less clear than in the sequential case. There is no analog of [8] (yet), and the world of parallel and distributed algorithms seems much more difficult to survey completely than the world of sequential algorithms. (Distinguishing between "parallel" and "distributed" in the sense of [7], it seems that an analog of [8] is within reach for parallel algorithms but not for distributed ones.)

A particular headache that has come up several times in my discussions with Yuri Gurevich is the question of cumulative updates. As a simple example, suppose we want to count the number of elements in some finite universe U (in an ASM). The natural way to do this is to have a counter, initialized to zero, which each of the relevant elements then increments by one. The problem is to do this in parallel, without arbitrarily fixing an order in which the various elements are to act. The rule

```
do for all v with U(v)
   c := c + 1
enddo
```

only increments the counter by one, no matter how many elements are in U. If we have an ordering of these elements, then we can let each one increment the counter in turn. But without an ordering, a genuinely parallel version of the algorithm seems to require something new.

The fundamental problem, though, isn't whether this or that situation calls for an extension of the framework but rather whether the framework can be completed at all. I am inclined to be optimistic: A few additions should cover all parallel algorithms. But I realize that my optimism may be the result of an overly naïve picture of the wild world of parallelism.

5 How Basic Are Sets?

The title of this section is intended to refer to the computational (ASM) side of the picture, not the pure mathematical (ZFC) side. Of course, sets are basic in ZFC, but this is to some extent the result of an arbitrary choice. Indeed, Cantor, the father of set theory, seems to have regarded sets as intrinsically equipped with an ordering, so perhaps some sort of (generalized) lists would be more fundamental.

On the computational side, too, there is a tendency to view interdefinable concepts, like (finite) sets and lists, as having equal claim to being fundamental. But it is shown in [3] that this tendency is not always appropriate. When we do not allow arbitrary choices (or, equivalently, an ordering) but do allow parallelism, and when we impose polynomial bounds on the total computation time of all processors, then a set-theoretic environment as in [2] is strictly stronger than one using tuples or lists instead. (The polynomial time bound is essential here. Without it, parallelism could compensate for the absence of an ordering by exploring all orderings.)

This result suggests to me that a set-based environment is more appropriate for this sort of computation than the more familiar list-based environments. There is some tension here with the fact that lists can be more straightforwardly implemented than sets. But the sort of computation under consideration, with no ordering allowed, is already intrinsically removed from implementation, where an ordering is implicitly given by the machine representation of elements.

These considerations raise a further question concerning choiceless polynomial time computation: Might some other environment be even better than the set-based one? Perhaps the considerations at the end of the preceding section, about cumulative updates, suggest an answer, namely to use multisets and allow cumulative updating of multiplicities of membership in a multiset. In other words, allow updates of the form "throw element x into multiset y," with the convention that, if several parallel processes execute this update (with the same x and y), then the result is that the multiplicity of x's membership in y is increased by the number of these processes.

But can one do better yet? Might there even be entirely new data structures that are particularly useful in the choiceless context?

6 Unidentified Objects

In this final section, I'd like to comment on an issue that comes up in many contexts, throughout computer science and mathematics, but is almost always ignored because it's trivial. My main point is that, if it's so trivial, we should be able to give a clear explanation very close to our trivializing intuition.

The issue arises in the theory of ASM's in connection with the importing or creating of new elements. It doesn't matter what the new element is, as long as it's new. So we refuse to worry about the exact choice of the new element. (If two or more elements are to be imported simultaneously, then we make sure they're distinct, but beyond this avoidance of "clashes" we refuse to worry.) A mathematician's immediate reaction is that we work, not with a particular structure in which a particular element has been imported, but with an isomorphism class of structures, a different but isomorphic structure for each choice of imported element. This works, but it doesn't quite correspond to intuition; intuition is still dealing with one structure, not a whole class of them. In some sense, intuition deals with a "generic" member of the isomorphism class, but what exactly (mathematically) is that?

The same issue and the same mathematical solution occur in connection with the syntax of just about any formalized language. One uses bound variables, but one doesn't care what particular variable is used (as long as clashes are avoided). It has become customary to annihilate the issue by saying "we work modulo α-conversion," meaning that expressions differing only in the choice of bound variables are to be identified. This amounts to the "isomorphism class" viewpoint of the preceding paragraph. Another approach was taken by Bourbaki [4]. Their official syntax for logic and set theory (and thus for all of mathematics) has no bound variables. In their places are occurrences of the symbol $\square$. To

indicate which occurrences correspond to the same variable, these are joined to each other (and to the binding operator) by lines. This seems to me to be closer to intuition, but rather unreadable. And it applies only to syntactic situations; I don't see how to adapt it to a semantic situation like ASM's. (I find it somewhat reassuring that the highly respected mathematicians of Bourbaki — or at least some of them — found this issue to be worth thinking about to the extent of producing a rather unorthodox solution.)

Another approach to this issue, in the syntactic context of bound variables, is the use of de Bruijn indices. These indices link a variable-binder, such as a quantifier, with the occurrences of the variables it binds, by specifying the difference between them in quantifier depth. That is, in place of a Bourbaki box with a line joining it to a quantifier, one would have a number indicating that that the relevant quantifier is to be found so and so many levels less deep in the parse tree. This notation strikes me as farther from intuition than Bourbaki's boxes but closer than isomorphism classes. In terms of human readability, it seems no better than the boxes, but I understand computers read it quite well. (Perhaps we humans just need to be wired differently.) I see no way to adapt this approach to non-syntactic situations, like the choice of new elements created in an ASM.

A variant of the isomorphism class approach is suggested by the topos-theoretic view of generic or variable objects [10]. In the present context, this amounts to regarding the isomorphism class not simply as a class but as an indexed or parametrized family, the parameters being the individual choices. Thus, for instance, an ASM that imports two elements would be viewed as a family of ASM's indexed by ordered pairs of distinct elements. The ASM indexed by the pair (x, y) is the one in which first x and then y were imported. An elegant framework and notation for this approach (applied to bound variables) is given in [6].

The issue of unidentified objects arises in many other contexts. Indeed, one can claim that the exact identity of mathematical objects never matters; all one cares about is the structure relating them. For example, it never matters whether real numbers are Dedekind cuts or equivalence classes of Cauchy sequences, as long as they form a complete ordered field. The following quotation from [10, page 119], though intended to emphasize the topos theorist's view of sets in contrast to the set theorist's cumulative hierarchy, seems to accurately describe how most mathematicians view sets.

> An abstract set X has elements each of which has no internal structure whatsoever; X has no internal structure except for equality and inequality of pairs of elements, and has no external properties save its cardinality; still an abstract set is more refined (less abstract) than a cardinal number in that it does have elements while a cardinal number does not.

This comes very close to saying that the actual elements don't matter as long as there are some (and equality is determined). But I claim that we have no

entirely satisfactory semantics for dealing with the concept of an arbitrary object whose actual identity doesn't matter (but whose distinctness from other arbitrary objects may be important).

Notice that the problem I am raising is a semantical one. Syntactically, we can handle objects whose identity doesn't matter: just add constant symbols to the language to serve as names for the desired objects. This process is well understood in the context of first-order logic (see for example [15, Chapter 4]; it plays a crucial role in the proof of Gödel's completeness theorem). But passing from syntax to semantics requires choosing elements to serve as the denotations of the new constant symbols, and that brings us right back to the original problem: We need to choose an element but without caring which element it is.

Finally, let me mention that this problem threatens to become more acute if one considers quantum computation or indeed any quantum phenomena. In the quantum world, actual physical objects, like elementary particles, have identities in only a very limited sense. It makes sense to talk about an electron and another electron, but not to talk about this electron and that one — interchanging the two electrons in some physical process yields not another process but another channel for the same process, and two channels may interfere (constructively or destructively).

A Appendix on Defaults and Environment

In this appendix, we collect some observations and questions about ASM's whose connection to pure mathematics is even more tenuous than in the last section. Indeed, most of them are based on a contrast rather than a connection between the computational and mathematical worlds.

A major difference between the viewpoints of ASM's (together with most of computer science) and pure mathematics is that in the former a dynamic aspect is always present. An ASM doesn't just sit there, it undergoes transitions — as the old name "evolving algebra" emphasizes. Of course, pure mathematics is also capable of dealing with dynamic situations, but this is always explicitly emphasized, not part of a universal background as in ASM's.

There is more to the dynamic aspect of ASM's than just that their dynamic functions can change their values. It is also important that dynamic functions retain their previous values unless explicitly changed. Ironically, this apparently simple default assumption, favoring persistence of values, can increase the logical complexity of what an ASM does. For example, the action of a parallel rule

```
do for all v
   R(v)
enddo
```

could be described in existential logic (or in the existential fixed point logic advocated in [1]) provided its body $R(v)$ could be so described. The parallel rule executes an update if and only if *there exists* a value of v for which $R(v)$ executes that update. But the next state cannot be described existentially, because the

inaction of the rule, the persistence of dynamic functions when not updated, requires a universal quantifier for its description.

The fact that parallel ASM's can be limited to quantifier-free guards, using `do for all` to simulate quantifiers, is ultimately due to the presence of an implicit universal quantifier in the default requirement that values persist unless explicitly changed. (In the last two paragraphs, I've pretended for simplicity that the ASM's under consideration never attempt conflicting updates. The possibility of clashes would introduce additional universal quantifiers, because an update is executed if some $R(v)$ would execute it (ignoring clashes) and *there are no* w_1 and w_2 for which $R(w_1)$ and $R(w_2)$ would execute conflicting updates.)

Another sort of default adds complexity not in the logic but in the computational aspects of ASM's. This default is the assumption that, when new elements are created (or imported from the reserve), they arrive with no structure: All Boolean functions, except equality, produce `false` when one of their arguments is a freshly created element; all non-Boolean functions produce `undef` under these circumstances. Thus, for example, if P is a binary Boolean function and x is a newly created element, then $P(x, y)$ miraculously has the value `false` for all y. If we ask how it got that value, there are several possible viewpoints. One is that creating an element really means importing it from the reserve, and that the appropriate default values were already there while x was in the reserve — so the default value of $P(x, y)$ for newly imported x is just a matter of the persistence default discussed above. But of course this viewpoint requires that the initial state of a computation include the appropriate default values for reserve elements, an assumption that is appropriate only at a sufficiently high level of abstraction. At a lower level, one would have to ask how this initialization is to be performed. Another viewpoint is that the defaults are (at least implicitly) set by the ASM at the time the new element is created. This amounts to a fairly large scale parallel operation, not available in the sequential situation; it may be the approach closest to what happens in actual computers when new memory locations are allocated to a computation. A third viewpoint is that the setting of the defaults is, like the choice of which element to import, the responsibility of the environment. This seems to be the simplest approach, but it strikes me as a bit unfair to the environment. Making the environment choose the new element is reasonable, because this cannot be accomplished algorithmically (unless an ordering or some similar structure is available); but setting the defaults could be done algorithmically (in the case of parallel ASM's) so the justification for turning the job over to the environment seems to be only that it's more work than the ASM wants to do.

Let me close with a brief comment, related to the preceding only because it's about the environment. The environment of an ASM is used to model a variety of things: the choice of elements to import (or create), the arbitrary choices involved in non-determinism, input-output operations, and, in distributed computing, all the agents other than the one under consideration. How similar are these, really? There is of course one similarity, which caused them to all be called "environment" in the first place: They are not part of the algorithm (ASM)

under consideration, but they interact with it. Is there further similarity among some (or all) of these aspects of the environment? Are there *useful* distinctions to be made? ("Useful" means, at a minimum, more useful than just listing the various items as I did above.)

One rather imprecise but perhaps useful distinction is obtained by singling out those aspects of the environment's activity that one would expect to be included in a system for executing ASM programs. Such a system should not be expected to provide input or to perform the actions of distributed agents other than the one being simulated. But it could reasonably be expected to execute `import` rules on its own, or at most with the help of the operating system under which it runs.

References

1. Andreas Blass and Yuri Gurevich, "Existential fixed-point logic," in *Computation Theory and Logic*, ed. by E. Börger, Lecture Notes in Computer Science 270, Springer-Verlag (1987) 20–36.
2. Andreas Blass, Yuri Gurevich, and Saharon Shelah, "Choiceless polynomial time," *Ann. Pure Appl. Logic*, 100 (1999) 141–187.
3. Andreas Blass, Yuri Gurevich, and Jan Van den Bussche, "Abstract state machines and computationally complete query languages," this volume.
4. Nicolas Bourbaki, *Elements of Mathematics. Theory of Sets*, Hermann (1968).
5. Abraham Fraenkel, *Der Begriff "definit" und die Unabhängigkeit des Auswahlaxioms*, Sitzungsberichte der Preussischen Akademie der Wissenschaften, Physikalisch-Mathematische Klasse (1922) 253–257.
6. Murdoch J. Gabbay and Andrew M. Pitts, "A New Approach to Abstract Syntax Involving Binders," in *Proceedings 14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 1999*, IEEE Computer Society Press (1999) 214-224.
7. Yuri Gurevich, "Evolving Algebra 1993: Lipari Guide", in *Specification and Validation Methods*, ed. by E. Boerger, Oxford University Press, 1995, 9–36.
8. Yuri Gurevich, "Sequential abstract state machines capture sequential algorithms," *ACM Transactions on Computational Logic*, to appear.
9. Thomas Jech, *The Axiom of Choice*, North-Holland (1973).
10. F. William Lawvere, "Variable quantities and variable structures in topoi," in *Algebra, Topology, and Category Theory (A Collection of Papers in Honor of Samuel Eilenberg)*, ed. by A. Heller and M. Tierney, Academic Press (1976) 101–131.
11. Eugene C. Luschei, *The Logical Systems of Leśniewski*, North-Holland (1962).
12. Anthony P. Morse, *A Theory of Sets*, Academic Press (1965).
13. Anand Pillay, *Geometric Stability Theory*, Oxford University Press (1996).
14. Saharon Shelah, *Choiceless polynomial time logic: Inability to express*, paper number 634, to appear.
15. Joseph Shoenfield, *Mathematical Logic*, Addison-Wesley (1967).
16. Edward E. Slaminka, "A Brouwer translation theorem for free homeomorphisms," *Trans. Amer. Math. Soc.*, 306 (1988) 277–291.
17. Alfred Tarski, Andrzej Mostowski, and Abraham Robinson, *Undecidable Theories*, North-Holland (1953).

Abstract State Machines and Computationally Complete Query Languages

Andreas Blass[1], Yuri Gurevich[2], and Jan Van den Bussche[3]

[1] University of Michigan, USA
ablass@math.lsa.umich.edu
[2] Microsoft Research and
University of Michigan, USA
gurevich@microsoft.com
[3] Limburg University, Belgium
jan.vandenbussche@luc.ac.be

Abstract. Abstract state machines (ASMs) form a relatively new computation model holding the promise that they can simulate any computational system in lockstep. In particular, an instance of the ASM model has recently been introduced for computing queries to relational databases. This model, to which we refer as the BGS model, provides a powerful query language in which all computable queries can be expressed. In this paper, we show that when one is only interested in polynomial-time computations, BGS is strictly more powerful than both QL and $while_{new}$, two well-known computationally complete query languages. We then show that when a language such as $while_{new}$ is extended with a duplicate elimination mechanism, polynomial-time simulations between the language and BGS become possible.

1 Introduction

Abstract state machines (ASMs) were introduced as a new computation model, accompanied by the "ASM thesis" stating that any algorithm, or more broadly, any computational system, at any level of abstraction, can be simulated in lockstep by an ASM [7,13,14,15]. Recently, Blass, Gurevich, and Shelah (BGS) introduced an instance of the ASM model for expressing queries to relational databases [8].

Roughly, a BGS program is a complex rule, changing the values of certain dynamic functions at various arguments during the run of the program. Rules are built up from elementary updates by conditionals and parallel composition. The program is iterated until a halting condition is reached. A powerful sublanguage of terms provides set-theoretic operations on arbitrarily nested sets over the input data elements. Once "activated," these sets are incorporated in the run of the program, and can become arguments and values of dynamic functions. While any computable query can be expressed in BGS, the actual motivation of BGS to introduce their model was to study the complexity class denoted by $\widetilde{\text{C}}$PTIME, corresponding to BGS programs under a polynomial time restriction.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 22–33, 2000.

Computationally complete query languages have been known in database theory for some years now [1], and complexity classes similar to $\widetilde{\text{C}}$PTIME, denoted by GEN-PTIME and GEN-PSPACE, were introduced by Abiteboul and Vianu [6]. These classes can be defined in terms of the language $while_{new}$.[1] This language is the extension of first-order logic with the following features: (1) assignment to relation variables; (2) sequential composition; (3) while-loops; and (4) the introduction of new data elements in terms of tuples of existing ones. All computable queries can be expressed in $while_{new}$. The complexity classes GEN-PSPACE and GEN-PTIME are obtained by putting polynomial space and time restrictions on $while_{new}$ programs. Abiteboul and Vianu illustrated the effect of such restrictions by showing that under a polynomial space restriction, $while_{new}$ programs can no longer check the parity of the cardinality of a set.

The advent of the BGS model thus raises the natural question: how does $\widetilde{\text{C}}$PTIME compare to GEN-PTIME? We will show that $\widetilde{\text{C}}$PTIME is strictly stronger than GEN-PTIME, in the sense that there are classes of structures that can be separated in $\widetilde{\text{C}}$PTIME but not in GEN-PSPACE (and hence neither in GEN-PTIME).[2] We also identify the reason for this inequality: $while_{new}$ only has *tuple-based invention:* new data elements can only be introduced in terms of tuples of existing ones. By repeated application of tuple-based invention one can construct arbitrary lists. BGS, on the other hand, allowing the construction of arbitrary sets, also has a form of *set-based invention*. In the absence of an order on the data elements, it is impossible to simulate sets (which are unordered) using lists (which are ordered) without introducing a lot of duplication.

Our result should be correctly compared to what is known from the theory of object-creating query languages. It is already known [18] that set-based invention cannot be expressed in $while_{new}$. However, this is a statement about object-creating queries where invention is not merely a tool to give more power to query languages, but where we really want to see the new data elements in the result of the query. When only considering standard domain-preserving, or even just boolean queries, set-based invention seemed less relevant because for such queries $while_{new}$ is already complete. Our results show that set-based invention is still relevant, but we have to take complexity into account to see it.

When $while_{new}$ is extended with set-based invention, we show that the language obtained, denoted by $while^{sets}_{new}$, becomes polynomial-time equivalent with BGS (in a sense that will be made precise). Our work is thus related to the update language detTL for relational databases, introduced by Abiteboul and Vianu [3,5]. Some of the spirit of the ASM model (of which BGS is an instance) is clearly present in detTL, and the equivalence between detTL and $while_{new}$

[1] Abiteboul and Vianu used the name $while^{invent}$ in their paper [6], but use the name $while_{new}$ in their book with Hull [1], so we use the latter name.

[2] A program *separates* two classes K_0 and K_1 if it outputs 'false' on all structures in K_0 and 'true' on all structures in K_1.

seems to go without saying.[3] New to our result are the programming with sets and the added focus on polynomial time.

We conclude this introduction by mentioning some other related work. The very first computationally complete query language was QL, introduced by Chandra and Harel [9]. Because QL can be simulated in $while_{new}$ with only a polynomial time overhead [6,19], our negative result concerning $while_{new}$ applies as well to QL. We also should note that the well-known object-creating query language IQL, introduced by Abiteboul and Kanellakis [2], was set in a complex-object data model with set values, where the distinction between tuples and sets is blurred as one can always have a tuple with a set as a component. Indeed, IQL is polynomial-time equivalent to $while_{new}^{sets}$ [19] and thus also to BGS. Finally, we point out that interest in object creation in query languages has recently resurged in the context of Web databases [12]. Current proposals in this field introduce new data elements by constructing terms, and thus essentially employ tuple-based invention.

2 Preliminaries

A relational database scheme is modeled by a finite relational vocabulary, in the sense of mathematical logic [11], i.e., a finite set of relation names with associated arities. A relational database over a scheme Υ is modeled by a finite structure B over Υ, i.e., a finite domain D and, for each relation name $R \in \Upsilon$, a relation $R^B \subseteq D^r$, where r is the arity of R. The reader is assumed to be familiar with the syntax of first-order logic formulas over Υ, and the notion of truth of a formula φ in a structure B.

We next briefly describe the languages $while_{new}$, $while_{new}^{sets}$, and BGS. For full details we refer to the literature [1,8,19].

2.1 The Language $while_{new}$

An *FO statement* is any expression of the form

$$X := \{(x_1, \ldots, x_j) \mid \varphi\}$$

where X is a j-ary relation name, and $\varphi(x_1, \ldots, x_j)$ is a first-order formula. A *tuple-new statement* is any expression of the form

$$Y := \textbf{tup-new}\{(x_1, \ldots, x_j) \mid \varphi\}$$

where Y is a relation name of arity $j+1$, and φ is as before.

Programs in the language $while_{new}$ are now defined as follows: FO statements and tuple-new statements are programs; if Π_1 and Π_2 are programs, then so is

[3] To witness, in their book with Hull [1], Abiteboul and Vianu refer to their paper on detTL [5] as the original source for the language $while_{new}$, although no language in the style of $while_{new}$ is discussed in that paper.

their composition $\Pi_1; \Pi_2$; and if Π is a program and φ is a first-order sentence, then the while-loop **while** φ **do** Π **od** is a program.

Let Π be a program, let Υ be the vocabulary consisting of all the relation names mentioned in Π, and let A be a finite Υ-structure. The *result of applying* Π *to* A, denoted by $\Pi(A)$, is the Υ-structure defined as follows:

- If Π is the FO statement $X := \{(x_1, \ldots, x_j) \mid \varphi\}$, then $\Pi(A)$ equals A except for the interpretation of X, which is replaced by
$$\{(a_1, \ldots, a_j) \in A^j \mid A \models \varphi(a_1, \ldots, a_j)\}. \qquad (*)$$
- If Π is the tuple-new statement $Y :=$ **tup-new**$\{(x_1, \ldots, x_j) \mid \varphi\}$, then $\Pi(A)$ equals A in the interpretation of every relation name other than Y. The domain of $\Pi(A)$ is that of A, extended with as many new elements as there are tuples in the above set $(*)$. Let ι be an arbitrary bijection between the set $(*)$ and these new elements. Then the interpretation of Y in $\Pi(A)$ is defined as
$$\{(\bar{a}, \iota(\bar{a})) \mid A \models \varphi(\bar{a})\}.$$
- If Π is of the form $\Pi_1; \Pi_2$ then $\Pi(A)$ equals $\Pi_2(\Pi_1(A))$.
- If Π is of the form **while** φ **do** $\bar{\Pi}$ **od**, then $\Pi(A)$ equals $\bar{\Pi}^n(A)$, where n is the smallest natural number such that $\bar{\Pi}^n(A) \not\models \varphi$. If such a number does not exist, then $\Pi(A)$ is undefined (the while-loop does not terminate).

By the semantics of tuple-new statements (second item), $\Pi(A)$ is clearly defined up to A-isomorphism only (isomorphisms that leave A pointwise fixed). This is OK, because the particular choice of the newly invented domain elements really does not matter to us. When doing a complexity analysis, we will assume that the domain of A is an initial segment of the natural numbers, and that a tuple-new statement simply extends this initial segment.

When Υ_0 is a subset of Υ, and A is an Υ_0-structure, we can view A also as an Υ-structure by setting $A(X)$ empty for every relation name X in Υ not in Υ_0. In this way we can also talk about $\Pi(A)$. This convention formalizes the intuition of initializing relation names not part of the vocabulary of the input structure to the empty set. These relation names are used by the program as variables to do its computation and to contain its final output.

2.2 The Language *while*

The sublanguage obtained from $while_{new}$ by disallowing tuple-new statements is called *while* and has been extensively studied [1,10]. In finite model theory, the language *while* is better known under the equivalent form of first-order logic extended with the partial fixpoint operator [4].

2.3 The Language $while^{sets}_{new}$

A *set-new statement* is an expression of the form
$$Y := \textbf{set-new}\{(x, y) \mid \varphi\},$$

where Y is a binary relation name, and $\varphi(x, y)$ is a first-order formula.

The result $\Pi(A)$ of applying this set-new statement Π to a structure A, equals A in the interpretations of every relation name other than Y. In order to define the domain of $\Pi(A)$ and its interpretation of Y, consider the binary relation

$$S = \{(a, b) \in A^2 \mid A \models \varphi(a, b)\}.$$

We can view this relation as a set-valued function in the canonical way: for any a in the first column of S, $S(a) := \{b \mid (a, b) \in S\}$.[4] Now the domain of $\Pi(A)$ is that of A, extended with as many new elements as there are *different* sets in the range of S. Let ι be an arbitrary bijection between the range of S and these new elements. Then the interpretation of Y in $\Pi(A)$ is defined as

$$\{(a, \iota(S(a))) \mid \exists b : S(a, b)\}.$$

For example, the result of applying

$$Y := \textbf{set-new}\{(x, y) \mid E(x, y)\}$$

to the structure with domain $\{1, 2, 3\}$ where E equals

$$\{(1, 1), (1, 2), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3)\},$$

is the structure with domain $\{1, 2, 3, 4, 5\}$ where Y equals

$$\{(1, 4), (2, 4), (3, 5)\}.$$

By adding set-new statements to the language $while_{new}$, we obtain the language $while_{new}^{sets}$.

2.4 The BGS Model

BGS takes a functional point of view: computing means updating the values of certain user-defined, named, "dynamic" functions at various arguments. Arguments and values can be elements of the domain D of the input structure, as well as hereditarily finite sets built over D during the execution of the program. Formally, the set $\mathrm{HF}(D)$ of *hereditarily finite sets over* D is the smallest set such that if $x_1, \ldots, x_n$ are in $D \cup \mathrm{HF}(D)$, then $\{x_1, \ldots, x_n\}$ is in $\mathrm{HF}(D)$. Every dynamic function name has an associated arity r, and thus has, at any stage of the computation, an interpretation (which can be updated in later stages) as a function from $(D \cup \mathrm{HF}(D))^r$ to $D \cup \mathrm{HF}(D)$. The *extent* of such a function f is the set $\{(\bar{x}, f(\bar{x})) \mid \bar{x} \in (D \cup \mathrm{HF}(D))^r \text{ and } f(\bar{x}) \neq \emptyset\}$. At any stage of the computation, the extent of the interpretation of any dynamic function will be finite.

A number of *static* functions, which cannot be updated, are predefined: The relations of the input structure are given as boolean functions. The usual logical

[4] In SQL terminology this corresponds to grouping by the first column.

constants[5] and functions (true, false, and, or, not, equality) are provided. Finally, some set-theoretic constants and functions are provided: the empty set; the input domain; set membership; set union; singleton extraction, and pairing. The input domain is called 'Atoms'. Union is unary, working on a set of sets.

Terms can now be built up from variables, constants, functions, and the set constructor $\{t : v \in r : g\}$, where v is a variable that does not occur free in term r but can occur free in term t and boolean term g. Variable v becomes bound by the set constructor. The semantics is the obvious one of $\{t : v \in r \text{ and } g\}$.

Finally, *rules* express transitions between states by updating the dynamic functions. *Elementary update rules* are of the form $f(t_1, \ldots, t_j) := t_0$, where f is a dynamic function name (of arity j) and $t_1, \ldots, t_j$ are terms. The semantics is obvious. From elementary update rules more complex rules can be built by conditionals and parallel composition. More specifically:

- If g is a boolean term and R_1 and R_2 are rules, then so is **if** g **then** R_1 **else** R_2 **endif**, again with the obvious semantics.
- If v is a variable, r is a term in which v does not occur free, and R_0 is a rule in which v can occur free, then **forall** $v \in r$ **do** R_0 **enddo** is a rule in which v becomes bound. The semantics is to perform R_0 in parallel for all $v \in r$, except if this yields conflicting updates in which case we do nothing.

A *BGS program* now is simply a rule without free variables. A program Π is started in the initial state, where all dynamic functions have the empty extent, and all static functions are initialized by the input structure I. In a run of the program, successive states are computed, until the dynamic boolean constant 'Halt' (which is present in all programs) becomes true. The final state is then the result $\Pi(I)$. As with $while_{new}$ programs, a BGS program may not terminate on some inputs.

2.5 Examples

An example of a $while_{new}$ program is shown in Figure 1, and an example of a BGS program is shown in Figure 2. Both example programs work on directed graphs, modeled as structures whose domain is the set of nodes and which have a binary relation E holding the edges. Both programs compute, for all pairs of nodes (x, y), all shortest paths from x to y. They do not follow exactly the same algorithm; the $while_{new}$ program does a single-source single-target search in parallel for all source-target pairs (x, y), while the BGS program does a single-source all-targets search in parallel for all sources x.

In the $while_{new}$ program, a path $x_1 \ldots x_n$ is represented by invented values $p_1, \ldots, p_n$ such that the following relations, defined by the program, hold: $Path(x_1, x_n, p_i)$ for $i = 1, \ldots, n$; $Ref(p_i, x_i)$ for $i = 1, \ldots, n$; and $Child(p_i, p_{i+1})$ for $i = 1, \ldots, n-1$. The relations *Frontier* and X used in the program are auxiliary variables.

[5] As usual, constants are viewed as zero-ary functions.

$Path :=$ **tup-new**$\{(x, y) \mid x = y\}$;
$Ref := \{(p, x) \mid \exists y\, Path(x, y, p)\}$;
$Frontier := \{(x, y, p, z) \mid Path(x, y, p) \wedge E(x, z) \wedge z \neq x\}$;
while $Frontier \neq \emptyset$ **do**
  $X :=$ **tup-new**$\{(x, y, p, z) \mid Frontier(x, y, p, z)\}$;
  $Path := \{(x, y, q) \mid Path(x, y, q) \vee \exists p \exists z\, X(x, y, p, z, q)\}$;
  $Ref := \{(q, z) \mid Ref(q, z) \vee \exists x \exists y \exists p\, X(x, y, p, z, q)\}$;
  $Child := \{(p, q) \mid Child(p, q) \vee \exists x \exists y \exists z\, X(x, y, p, z, q)\}$;
  $Frontier := \{(x, y, q, z') \mid \exists p \exists z (X(x, y, p, z, q) \wedge z \neq y \wedge E(z, z'))\}$
od;
$Path := \{(x, y, p) \mid \exists p' (Path(x, y, p') \wedge Ref(p', y))\}$.

Fig. 1. A $while_{new}$ program computing all-pairs shortest paths.

if $Mode = 0$ **then**
  forall $x \in$ Atoms **do**
    $Reached(x) := \{x\}$,
    $Paths(x, x) := \{\{x\}\}$,
    $Frontier(x) := \{x\}$
  enddo,
  $Mode := 1$
endif,
if $Mode = 1$ **then**
  forall $x \in$ Atoms **do**
    $Old\_Frontier(x) := Frontier(x)$,
    $Frontier(x) := \{y : y \in \text{Atoms} : y \notin Reached(x)$
                and $\{z : z \in Frontier(x) : E(z, y)\} \neq \emptyset\}$
  enddo,
  $Mode := 2$
endif,
if $Mode = 2$ **then**
  forall $x \in$ Atoms **do**
    forall $y \in Frontier(x)$ **do**
      $Paths(x, y) := \{(p, y) :$
        $p \in \bigcup\{Paths(x, z) : z \in Old\_Frontier(x) : E(z, y)\} : \text{true}\}$
    enddo,
    $Reached(x) := Reached(x) \cup Frontier(x)$
  enddo,
  Halt $:= \bigcup\{Frontier(x) : x \in \text{Atoms} : \text{true}\} = \emptyset$,
  $Mode := 1$
endif.

Fig. 2. A BGS program computing all-pairs shortest paths.

In the BGS program, a path $x_1 \dots x_n$ is represented by a pair $(x_1 \dots x_{n-1}, x_n)$, where the $x_1 \dots x_{n-1}$ is again represented by a pair, recursively.[6] The base case $n = 1$ is represented by a singleton $\{x_1\}$. The program updates a dynamic binary function *Paths* such that $Paths(x, y)$ equals the set of shortest paths from x to y. Other dynamic functions and constants used by the program to aid the computation are *Mode*, *Reached*, *Frontier*, and *Old_Frontier*. The comma between rules denotes parallel composition, and is a shorthand for a trivial `forall do` construct. The natural numbers 0, 1, and 2 assigned to *Mode* are in $\mathrm{HF}(D)$ by their definition as von Neumann numerals: 0 is the empty set, and $n > 0$ is $\{0, \dots, n-1\}$, recursively [16]. The numbers 0 and 1 also play the role of the booleans false and true.

3 BGS and $while_{new}$ under Polynomial Time

In this section, we define what it means for two classes of structures over the same vocabulary to be separable in polynomial time by BGS programs, or by $while_{new}$ programs. We then prove that there exists a pair that is separable in polynomial time by a BGS program, but not by any $while_{new}$ program.

During the run of a BGS program on a structure with domain D, a certain number of sets in $\mathrm{HF}(D)$ are *activated,* meaning that at some point they appear in the extent of some dynamic function. Elements of active sets are also considered to be active, and this holds recursively. Similarly, during the run of a $while_{new}$ program on a structure, a certain number of new elements are invented. Activated sets and invented elements yield measures of space usage by BGS and $while_{new}$ programs, which are quite rough, but sufficient for our purposes. Equally rough measures of time spent by BGS and $while_{new}$ programs can be defined as follows: the time spent by a BGS program on a structure is the number of times the program is iterated until the halting condition is reached; the time spent by a $while_{new}$ program on a structure is the number of times an FO or tuple-new statement is executed during the run of the program.

In the following two paragraphs fix two disjoint classes K_0 and K_1 of structures over a common vocabulary.

Let Π be a BGS program using a boolean dynamic constant Output for output. We say that Π *separates* K_0 *from* K_1 if for any structure $A \in K_0 \cup K_1$, the value of Output in $\Pi(A)$ is false if $A \in K_0$, is true if $A \in K_1$. We say that Π *separates* K_0 *from* K_1 *in polynomial time* if moreover, there exist two polynomials $p(n)$ and $q(n)$ such that for any $A \in K_0 \cup K_1$, Π runs on A for at most $p(n)$ time, and activates at most $q(n)$ sets, where n is the cardinality of the domain of A.

Similarly, let Π be a $while_{new}$ program having some relation variable *Output*. We say that Π *separates* K_0 *from* K_1 if $\Pi(A)$ is defined for any structure $A \in K_0 \cup K_1$, and relation *Output* in $\Pi(A)$ is empty if $A \in K_0$, and is not empty if $A \in K_1$. We say that Π *separates* K_0 *from* K_1 *in polynomial time* if moreover,

[6] Recall that ordered pairs (x, y) are by definition in $\mathrm{HF}(D)$, as $\{\{x\}, \{x, y\}\}$ [16].

there exist two polynomials $p(n)$ and $q(n)$ such that for any $A \in K_0 \cup K_1$, Π runs on A for at most $p(n)$ time, and invents at most $q(n)$ elements, where n is the cardinality of the domain of A.

Since we do not care what the programs do on structures outside K_0 and K_1, the above notion of separation is quite liberal. Still, we will be able to obtain a negative result regarding the separating power of $while_{new}$ in polynomial time. Also, in our definition, it is important to polynomially restrict the space used as well as the time, because in BGS or $while_{new}$ it is possible to use an exponential amount of space even in an only linear amount of time.

We can prove (proof omitted):

Theorem 1. *There exist pairs of classes of structures that can be separated in polynomial time by a BGS program, but not by a $while_{new}$ program.*

Consider the vocabulary consisting of a single relation name P, which is unary. For any natural number n, define a structure I_n over this vocabulary as follows. The domain of I_n consists of 2^n elements. Exactly n of these satisfy the predicate P. The pair now for which we are going to prove the theorem was already considered by Blass, Gurevich and Shelah [8] and is the following: $K_0 = \{I_n \mid n \text{ even}\}$, and $K_1 = \{I_n \mid n \text{ odd}\}$. We can easily separate K_0 from K_1 by a BGS program in polynomial time: the program generates all subsets of P with even cardinality (which is in polynomial time because the cardinality of the input domain is 2^n), and then checks whether P itself was generated.

We can actually show that K_0 cannot be separated from K_1 by any $while_{new}$ program that can invent only a polynomial number of elements; the time spent by the program will be irrelevant.

Because of the equivalence between $while_{new}$ and the generic machine model of Abiteboul and Vianu [6], Theorem 1 implies that generic machines are strictly weaker than BGS in the context of polynomial time computation. This result corrects a tentative claim ('the simulation in the reverse direction can, it seems, be carried out using the "form and matter" considerations in Section 9') near the end of Section 1 of the BGS paper [8]. The form and matter considerations mentioned there involve tuples rather than sets as "matter" and therefore run into the same duplication problem as $while_{new}$.

4 Polynomial Time Equivalence of BGS and $\boldsymbol{while^{sets}_{new}}$

In this section, we formally define notions of polynomial-time simulation of BGS programs by $while^{sets}_{new}$ programs, and vice versa, and show that such simulations exist.

4.1 Simulating $\boldsymbol{while^{sets}_{new}}$ in BGS

To simulate $while^{sets}_{new}$ in BGS, we need some way to represent elements that are invented by a $while^{sets}_{new}$ program by hereditarily finite sets that can be constructed by a BGS program. For elements invented by a **tup-new** statement, we

already did this in the previous section, where we described a list-construction semantics for **tup-new**.[7] So it remains to describe a set-construction semantics for **set-new**.

To this end, recall the semantics of a set-new statement $Y :=$ **set-new** S on a structure A (where S is a binary relation on A defined by some first-order formula), which assigns to relation name Y the relation $\{(a, \iota(\varphi(a))) \mid \exists b : S(a,b)\}$ for some bijection ι from the range of S (viewed as a set-valued function) to new elements. We fix this bijection ι uniformly as follows. Assume it is the mth time we are performing a tuple-new or set-new statement in the execution of the program. Then $\iota(S(a))$ is defined to be the pair

$$(S(a), \lambda_m),$$

where λ_m is as defined in the previous section.

We now say that a BGS program Π' *simulates* a $while^{sets}_{new}$ program Π if for every input structure I, if $\Pi(I)$ is defined then so is $\Pi'(I)$, and for every relation variable X of Π, say of arity r, there is an r-ary boolean dynamic function $\widehat{X}$ of Π', such that the tuples in X in $\Pi(I)$ are exactly the tuples at which $\widehat{X}$ is true in $\Pi'(I)$. Moreover, we say that the simulation is *linear-step, polynomial-space* if there exists a constant c and a polynomial p such that for every input structure I where $\Pi(I)$ is defined, the following holds. Let the time for which Π runs on I be t, and let the number of invented elements during the run be s. Then Π' runs on I for at most ct time, activating at most $p(n+s)$ sets, where n is the cardinality of the domain of I.

Here, in close analogy to what we defined for $while_{new}$ programs at the beginning of Section 3, we define the time spent by a $while^{sets}_{new}$ program on a structure as the number of times an FO, tuple-new, or set-new statement is executed during the run of the program.

Note that, while we allow a polynomial overhead in space usage, we allow only a linear overhead in the running time of the simulation. A weaker notion of polynomial time simulation could be defined, allowing a polynomial overhead also for running time, but we will not need to consider this weaker notion as we will be able to obtain positive results for our stronger notion.

We can show (proof omitted):

Theorem 2. *Every $while^{sets}_{new}$ program can be linear-step, polynomial-space simulated by a BGS program.*

4.2 Simulating BGS in $while^{sets}_{new}$

To simulate BGS in $while^{sets}_{new}$, we need some way to represent hereditarily finite sets by invented elements. To this end, we observe that for any finite domain D, the structure $(D \cup \mathrm{HF}(D), \in)$ is an (infinite) directed acyclic graph. At any stage in the run of a BGS program on a structure with domain D, the active

[7] Lists are special kinds of sets: a list of length n is a mapping from $\{1, \ldots, n\}$ to the set of members of the list, and a mapping is a set of ordered pairs.

sets, together with the elements of D, generate a finite subgraph of this graph. The simulating $while^{sets}_{new}$ program will maintain a copy of that subgraph, where the active sets are represented by invented elements, and the elements of D are represented by themselves. The membership relation $\in$ will be stored in a relation variable *Epsilon*.

We now say that a $while^{sets}_{new}$ program Π' *simulates* a BGS program Π if for every input structure I, if $\Pi(I)$ is defined then so is $\Pi'(I)$, and for every dynamic function name f of Π, say of arity r, there is an $(r+1)$-ary relation variable $\widehat{f}$ of Π', such that $\widehat{f}$ in $\Pi'(I)$ equals exactly the extent of f in $\Pi(I)$, under a representation of the active hereditarily finite sets by invented elements as given in relation *Epsilon* in $\Pi'(I)$. Moreover, we say that the simulation is *linear-step, polynomial-space* if there exist a constant c and a polynomial p such that for every input structure I where $\Pi(I)$ is defined, the following holds. Let the time for which Π runs on I be t, and let the number of sets activated during the run be s. Then Π' runs on I for at most ct time, inventing at most $p(s)$ elements.[8]

We can show (proof omitted):

Theorem 3. *Every BGS program can be linear-step, polynomial-space simulated by a* $while^{sets}_{new}$ *program.*

Acknowledgment. Thanks go to Marc Spielmann for proofreading an earlier draft of this paper.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.
2. S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. *Journal of the ACM*, 45(5):798–842, 1998.
3. S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings 7th ACM Symposium on Principles of Database Systems*, pages 240–250, 1988.
4. S. Abiteboul and V. Vianu. Fixpoint extensions of first-order logic and Datalog-like languages. In *Proceedings Fourth Annual Symposium on Logic in Computer Science*, pages 71–79. IEEE Computer Society Press, 1989.
5. S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Sciences*, 41(2):181–229, 1990.
6. S. Abiteboul and V. Vianu. Generic computation and its complexity. In *Proceedings 23rd ACM Symposium on the Theory of Computing*, pages 209–219, 1991.

[8] The reader will have noticed that, while here we require that Π' invents at most $p(s)$ elements, in the notion of polynomial-space simulation of $while^{sets}_{new}$ programs by BGS programs as defined in the previous subsection, we allowed the simulating BGS program to activate $p(n+s)$ sets. The reason for this is that, even if a $while^{sets}_{new}$ program Π does not invent any new elements (i.e., $s = 0$), a simulating BGS program still needs to activate some sets just to evaluate the first-order formulas used in Π.

7. Abstract state machines Web pages. ⟨www.eecs.umich.edu/gasm⟩.
8. A. Blass, Y. Gurevich, and S. Shelah. Choiceless polynomial time. *Annals of Pure and Applied Logic*. To appear; also available from [7].
9. A. Chandra and D. Harel. Computable queries for relational data bases. *Journal of Computer and System Sciences*, 21(2):156–178, 1980.
10. H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1995.
11. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Mathematical Logic*. Undergraduate Texts in Mathematics. Springer-Verlag, 1984.
12. M. Fernández, D. Florescu, J. Kang, A. Levy, and D. Suciu. Catching the boat with Strudel: Experiences with a Web-site management system. *SIGMOD Record*, 27(2):414–425, 1998. Proceedings ACM SIGMOD International Conference on Management of Data.
13. Y. Gurevich. Evolving algebras: An attempt to discover semantics. *Bulletin of the European Association for Theoretical Computer Science*, 43:264–284, 1991.
14. Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
15. Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, EECS Department, 1997.
16. P. Halmos. *Naive Set Theory*. Van Nostrand Reinhold, 1960.
17. P.G. Kolaitis and M.Y. Vardi. Infinitary logics and 0-1 laws. *Information and Computation*, 98(2):258–294, 1992.
18. J. Van den Bussche and J. Paredaens. The expressive power of complex values in object-based data models. *Information and Computation*, 120:220–236, 1995.
19. J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating database transformation languages. *Journal of the ACM*, 44(2):272–319, 1997.

On Verification of Refinements of Timed Distributed Algorithms

J. Cohen[1] and A. Slissenko[1,2]

[1] UniversityParis-12
Dept. of Informatics
61 Av. du Gén. de Gaulle
94010, Créteil, France
{cohen,slissenko}@univ-paris12.fr
[2] St.Petersburg Institute for
Informatics and Automation of
Russian Academy of Sciences

Abstract. This work is an attempt to apply Gurevich Abstract State Machines methodology to the verification of refinements of real-time distributed asynchronous algorithms. We define refinements following the semantical framework of observability, however, with respect to chosen pieces of the program. The time we consider is continuous as our motivation is related to systems of control that are usually specified within continuous time framework; the same framework is valid for discrete time. We remark that refinement of timed programs is not a simple replacement of a part of a program by making it more detailed. As an example to illustrate this we take Lamport's Bakery Algorithm with real-time constraints. However, one of the key questions related to the verification of refinements is the preservation of verification proofs for the non refined initial algorithm as much as possible when verifying the refinement. This is the case for the notion of refinement we define. We introduce a notion of asynchronous timed distributed algorithm, define its semantics and discuss in what logic can be expressed its functioning. Then we introduce notions of refinement, and consider a refinement of interprocess communication of real-time Lamport's Bakery algorithm using parallel message exchange. Such a refinement, contrary to our intuition, demands some non evident transformation of the initial algorithm to make it correct.

1 Introduction

The goal of this paper is to define asynchronous timed algorithms and their refinements. To illustrate our system of notions we study a concrete algorithm, namely a real-time version of Lamport's Bakery, within this framework. (In fact, the notions we introduce, are intended to model more complex algorithms, composed from many modules with non trivial asynchronous interaction.) Though the classical notion of asynchronous algorithm does not have any metric time constraints, practical implementation of such an algorithm usually cannot have,

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 34–49, 2000.

say, unlimited delays, and to be practical, some time constraints may be necessary, at least at some level of refinement of such an algorithm. Moreover, the time is very intuitive, and this is one of the reasons why time arguments are largely used in reasoning about distributed algorithms.

What is more important concerning the time, is that it often appears in initial requirements specification. So if we rewrite *directly* the initial specification using time one can hardly require formal justification of this passage, but if we *eliminate* time then, clearly, we *modify* the initial specification, and this modification is to be justified. But formal justification will demand an adequate notion of time to express what we are modifying. For example, Lamport in [7] writes, giving a requirements specification of the critical section problem : "*At any time,* at most one computer may be in its critical section". Clearly, we can directly rewrite it as $\forall\, t\, (CS(t,p) \rightarrow \forall\, q \neq p \neg CS(t,q))$, where t is a time variable, p and q are variables for processes and $CS(t,r)$ is a predicate which states that "process r is in critical section at the moment t". And that is how we proceed here.

The paper [2] by E. Börger, Yu. Gurevich and D. Rozenzweig was a starting point of our work. We use the methodology of Gurevich Abstract State Machines (GASM) and not the exact notion from, say, [6]. More precisely, we assemble in the style of GASM a minimal notion sufficient for our goal from simple programming constructions starting from assignments as used in GASM. The GASM methodology gives basic principles how to define semantics of such algorithms. To be adequate to Lamport's text [7] we introduce asynchronous executions of our distributed algorithms with delays. To give to the algorithm a real-time flavor we consider here restricted delays, and thus, slightly deviate from the original Lamport's Bakery.

Within our semantics the Lamport's Bakery algorithm rewritten in our system of notations can be proved to be correct following Lamport's original proof.

However, our main goal is to define a notion of refinement of practical significance and to study whether the verification proof for the non-refined initial algorithm can be preserved for the verification of the refined one. In our definition of refinement we follow the idea of observational equivalence related to some structural properties of the program. And for this notion the proof preservation mentioned above takes place. Constructing a correct refinement remains a non-trivial problem, and we illustrate it by giving a refinement of communications in real-time Lamport's Bakery. We also remark that a straightforward 'local' replacement of operators that we wish to refine by their refinements does not give a correct refinement.

Though the notion of refinement was studied in many papers (e. g. see [9,8,3,1,5]), our notion treats real-time algorithms, where the time can be used without any restrictions and the refinement itself is being done with respect to chosen pieces of the program. These properties are motivated by the analysis of practical refinements and the verification problem, and all this differs our notion from the notions of refinement considered in other papers. For example, in the paper [8] the time is represented only as ticks. The properties of such systems are

described in RTTL (Real-time Temporal Logic) where concrete bounds on the number of ticks are admissible as real-time constraints. Treating refinement as an observational equivalence the author proves a theorem about preservation of properties under this equivalence. The limited formalisms of [8] may have some algorithmic advantages, but we look for formalisms that permit to express much richer properties.

The analysis of refinements and refinement mappings in [1] defined in terms of runs of state machines touches many subtle questions in a semantical framework. A state machines framework with many compositionality properties is also developed in [5] aimed at compiler construction.

Paper [3], which gives many references on the study of the refinement, is concentrated on interactive systems and compositionality. There is no time explicitly involved. With respect to compositionality our case is different because of real-time constraints, and we have not yet studied this topic.

2 Asynchronous Distributed Algorithms and Their Timed Semantics

Introducing explicit time we pursue two goals: first, to give a unified and intuitive view on semantics and verification proofs and, second, to give possibility to introduce time constraints in specifications of algorithm when refining it or when making precisions of initial formulation.

2.1 Asynchronous Distributed Algorithm Definition

The vocabulary of a language to describe asynchronous distributed timed algorithms (further, simply "algorithms") consists of sorts and functions (predicates are treated as a particular case of functions). A part of the elements of the vocabulary will have a predefined interpretation, and the remaining elements will be called *abstract*. Among abstract functions there are distinguished *input* or *output* ones, the two sets being disjoint. A relation between input and output functions constitutes the main part of the specification of functioning of the system under consideration.

Sorts. The sort time $\mathcal{T}$ will be interpreted as $\mathbb{R}_{\geq 0}$, and will be a subsort of the sort of reals $\mathbb{R}$. The sort ∞ is interpreted as an element which is greater than all the elements of $\mathcal{T}$. The sort $\mathcal{T}^\infty$ is $\mathcal{T} \cup \{\infty\}$, and sort $\mathcal{T}$ is considered as subsort of $\mathcal{T}^\infty$. The sort *Bool* denotes Boolean values. There can be other sorts that are presumed to be abstract and finite. Each sort is supplied with a set of variables, these sets are disjoint for different sorts.

Functions. Pre-interpreted constants for reals are all $\mathbb{Q}$. Nullary function CT represents the current time. Relations $\{<, \leq\}$ for $\mathcal{T}$ and $\mathcal{T}^\infty$ are usual, and there is equality for each sort. Abstract functions (other than predicate of equality) can be of type $\mathcal{X} \to Z$, where $\mathcal{X}$ is a product of finite abstract sorts, and Z is an arbitrary sort.

The abstract functions are partitioned into *external* and *internal*. External ones are those that cannot be changed by algorithm, internal ones are those that

can be changed by algorithm. External functions that depend on time are *input* functions. Internal functions of zero arity, also called *nullary* functions, play the role of *identifiers*, as well as terms $f(X)$ for a fixed value of its arguments X. The mentioned arguments X can be viewed as parameters. Identifiers $f(X)$ and $f(Y)$ are different if $X \neq Y$.

Syntax.

$Term$ ::= $Variable \mid Function(Term_1, ..., Term_n)$,

here and in the next line $Function$ is an n-ary function symbol.

$InternalTerm$ ::= $InternalFunction(Term_1, ..., Term_n)$.

$Formula$ ::= standard first order formula.

$Guard$ ::= Formula without time or real variables. It must be either closed or may have free variables ω bounded by **ForAll** $\omega \in \Omega$, see below the definition of the operator.

$Assignment$ ::= $InternalTerm := Term$.

Asynchronous timed distributed algorithm over a vocabulary V is a pair of the form ($IniState$, $\{P_1, \ldots, P_N\}$), where $IniState$ is a formula defining initial state, i. e. the values of all the abstract functions of V (at time 0 if a function has a time argument), and each P_i is a *program* or *process* that is defined below. (We use the word "operator" where in programming languages one uses "statement" and where Yu. Gurevich [6] usually uses the word "rule".)

$Program$ ::= **Prog** $Operator$ **EndProg**

$Operator$ ::= $Assignment$ – defined above

| **Seq** $Operator; \ldots; Operator$ **EndSeq** – sequential block

| **Par** $Operator \| \ldots \| Operator$ **EndPar** – parallel block

| **If** $Guard$ **Then** $Operator$ **EndIf** – conditional operator

| **Repeat** $Operator$ **Until** $Guard$ **EndRepeat** – repeat operator

Extensions of the syntax like

(1) **Par ForAll** $\omega \in \Omega$ **Do** $Op_1(\omega) \| \ldots \| Op_k(\omega)$ **EndPar**, where Ω is a finite set, (2) **Repeat** $Operator$ **EndRepeat**, have obvious meaning.

2.2 Timed Semantics of Asynchronous Distributed Algorithms

The semantics we define presumes that the actions are *instantaneous*, but between consecutive actions there is a *delay*. Imposing different constraints on delays we can vary details of the semantics. We assume that delays are bounded and can be different for different operators. Not to complicate the situation too much, we suppose that there are two delays: one for "internal" operations of a process, and the other for "external" information exchange. That means that assignments or guards that use information from other processes have "external" delays. We assume that there are positive constants δ_{int} and δ_{ext} that give bounds on the mentioned delays: $t \leq_{int} t' \Leftrightarrow t < t' \leq (t + \delta_{int})$ and $t \leq_{ext} t' \Leftrightarrow t < t' \leq (t + \delta_{ext})$. The notation $t \leq_{delay} t'$ will mean that the delay between t and t' can be determined by syntactical criterion: the delay is internal if the concerned operator deals only with the identifiers of the same process. For example, an assignment of a process P is classified as external iff it is of the form $f(\overline{\eta}) := \theta'$, where f is an internal function of P, and either the terms of list $\overline{\eta}$ or term θ' contain functions of other processes. In this case we tacitly presume

that the process has to send a query to another processes to get the necessary information, that implies an external delay, and then to receive this information, that contributes again to the external delay. Similar for guards.

Remark that we can eliminate Zero runs if we impose a positive lower bound on delays.

The semantics is defined by a propagation of time moments of execution of operators and by defining simultaneously the values of internal functions. To simplify the presentation we will give first the rules of propagation of time though, strictly speaking, some of them involve results of function evaluation. Below *Prog* stands for *Program*, *Op* stands for *Operator*.

Time Propagation Rules.
Intuitive meaning of vertical arrows below is as follows: "$Op\downarrow(t)$" means "t is the termination time of the operator", "$\uparrow(t)Op$" means "t is the start time of the operator". Double arrow is used when the algorithm takes values or change values. So $\uparrow(t)$ goes down to an assignment, is changed to $\downarrow(t)$, and the latter goes up to the end of the operator. Symbol $\vdash$ means the expression at the left can be rewritten as an expression at the right of $\vdash$. Below t' is arbitrary moment such that $t' \geq_{int} t$, and t_0, t_1 are arbitrary moments such that $t_1 \geq_{delay} t_0 \geq_{delay} t$.

- Given an algorithm $(IniState, \{Prog_1, \ldots, Prog_N\})$ its *initial time distribution* is $\vdash \{\uparrow(0)Prog_1, \ldots, \uparrow(0)Prog_N\}$.
- The following transition says that the execution is finite and halts:
  $\{Prog_1\downarrow(\infty), \ldots, Prog_N\downarrow(\infty)\} \vdash \{Prog_1, \ldots, Prog_N\}\downarrow(\infty)$.
- $\uparrow(t)\mathbf{Prog} \vdash \mathbf{Prog}\uparrow(t')$. • $\downarrow(t)\mathbf{EndProg} \vdash \mathbf{EndProg}\downarrow(\infty)$.
- $\uparrow(t)\mathbf{Seq} \vdash \mathbf{Seq}\uparrow(t')$. • $\downarrow(t); \vdash ; \uparrow(t')$. • $\downarrow(t)\mathbf{EndSeq} \vdash \mathbf{EndSeq}\downarrow(t')$.
- $\uparrow(t)\mathbf{Par}\, Op_1 \| \ldots \| Op_n\, \mathbf{EndPar} \vdash \mathbf{Par}\ \uparrow(t_1)Op_1 \| \ldots \| \uparrow(t_n)Op_n\, \mathbf{EndPar}$
  for any $t_1, \ldots, t_n \geq_{int} t$.
- $\mathbf{Par}\, Op_1\downarrow(t_1) \| \ldots \| Op_n\downarrow(t_n)\, \mathbf{EndPar}$
  $\vdash \mathbf{Par}\, Op_1 \| \ldots \| Op_n\, \mathbf{EndPar}\downarrow(t')$ for any $t' \geq_{int} \max\{t_1, \ldots, t_n\}$.
- $\uparrow(t)\mathbf{If} \vdash \mathbf{If}\uparrow(t')$. • $\downarrow(t)\mathbf{EndIf} \vdash \mathbf{EndIf}\downarrow(t')$.
- $\mathbf{If}\ \uparrow(t)Guard\, \mathbf{Then}\, Op\, \mathbf{EndIf} \vdash \mathbf{If}\, Guard\, \Uparrow(t_0)\, \mathbf{Then}\, Op\, \mathbf{EndIf}$.
- $\mathbf{If}\, Guard\, \Uparrow(t)\, \mathbf{Then}\, Op\, \mathbf{EndIf}$
  $\vdash \begin{cases} \mathbf{If}\, Guard\, \mathbf{Then}\ \uparrow(t_0)Op\, \mathbf{EndIf} & \text{if } Guard(t) \text{ is true,} \\ \mathbf{If}\, Guard\, \mathbf{Then}\, Op\, \mathbf{EndIf}\downarrow(t_0) & \text{otherwise.} \end{cases}$
- $\uparrow(t)\mathbf{Repeat} \vdash \mathbf{Repeat}\uparrow(t')$. • $\downarrow(t)\mathbf{Until} \vdash \mathbf{Until}\uparrow(t')$.
- $\mathbf{Repeat}\, Op\, \mathbf{Until}\ \uparrow(t)Guard\, \mathbf{EndRepeat}$
  $\vdash \mathbf{Repeat}\, Op\, \mathbf{Until}\, Guard\, \Uparrow(t_0)\, \mathbf{EndRepeat}$.
- $\mathbf{Repeat}\, Op\, \mathbf{Until}\, Guard\, \Uparrow(t)\, \mathbf{EndRepeat}$
  $\vdash \begin{cases} \mathbf{Repeat}\uparrow(t_0)\, Op\, \mathbf{Until}\, Guard\, \mathbf{EndRepeat}, & \text{if } Guard(t) \text{ is false,} \\ \mathbf{Repeat}\, Op\, \mathbf{Until}\, Guard\, \mathbf{EndRepeat}\downarrow(t_0), & \text{otherwise.} \end{cases}$
- $\uparrow(t)\theta{:=}\theta' \vdash \theta\Uparrow(t_0){:=}\theta'$. • $\theta\Uparrow(t_0){:=}\theta' \vdash \theta{:=}\Downarrow(t_1)\theta'$.
- $\theta{:=}\Downarrow(t_1)\theta' \vdash \theta{:=}\theta'\downarrow(t')$.

Let $\theta = f(\eta_1, \ldots, \eta_n)$, where f is an internal function. To change the value of θ we are to find the values of arguments $\eta_1, \ldots, \eta_n$ of f and the value of θ' to assign. We take all these values at t_0 and *change* the value of θ at $t_1 \geq_{delay} t_0$.

Definition of Run (Semantics of Algorithms).
We consider arbitrary inputs. The internal functions in our definition below will be piecewise constant with pieces being left-open right-closed. Such a definition was elaborated for another type of GASM in [4].

Given an algorithm $\mathcal{A}$, an interpretation of sorts and values of inputs satisfying $IniState$ for $t = 0$, an execution of $\mathcal{A}$ for these inputs defines the values of its internal functions $\rho(t)$ at least for some prefix $\mathcal{T}_0$ of $\mathcal{T}$ that together with the inputs constitute a *complete* run $\{\widetilde{\rho}(t)\}_{t\in\mathcal{T}_0}$ of the algorithm. Together with ρ we will define other notions, either auxiliary or giving more detailed information on the execution of the algorithm to use them in the analysis of refinements.

Clearly, the run $\widetilde{\rho}(t)$ which is defined at t determines an interpretation of the vocabulary. We will denote by $f[\mathcal{I}]$, where $\mathcal{I}$ is an interpretation of a part of the vocabulary containing f, the interpretation of f given by $\mathcal{I}$. For example, $f[\rho(t)](X)$ or $f(X)[\rho(t)]$ is the value of $f(X)$ as defined by $\rho(t)$. We extend this notation on terms in a usual way. If we have an interpretation $\mathcal{I}(t)$ of V for every time moment we can define for each $f \in V : \mathcal{X} \to Z$ an interpretation $\mathcal{I}^\circ$ of its timed image $f^\circ : \mathcal{T} \times \mathcal{X} \to Z$ (symbol f° is in some other vocabulary V°) as $f^\circ(t, X)[\mathcal{I}^\circ] = f(X)[\mathcal{I}(t)]$. Consider an algorithm $\mathcal{A} = (IniState, \{P_1, \ldots, P_N\})$ over V. We assume that the initial state of $\mathcal{A}$ uniquely determines the values of internal functions at time 0 being given an input. Let the external functions be given and suppose that the algorithm starts its execution. Denote by $\rho(t)$ the values of internal functions at time t that are to be defined for $t > 0$. An internal function can be changed only by assignments. So we look for the moments of time at which the assignments are executed. We assume that all the guards and operators occurring in $\mathcal{A}$ are enumerated so that we can speak about their occurrences. By default, when mentioning a guard or an operator we mean its occurrence in $\mathcal{A}$.

Applying Time Propagation Rules to the algorithm $\mathcal{A}$, starting from initial time distribution, we get an expression that is a list of programs with occurrences of *arrow expressions* $\downarrow(t)$, $\uparrow(t)$, $\downdownarrows(t)$ and $\upuparrows(t)$ for concrete moments t. Such an occurrence of time will be called *arrowed occurrence*. An algorithm with arrow expressions will be called *timed algorithm expression (TAE)*. Remark that to make an assignment we take the values of all involved terms at the same point. But the changed value will be valid only after this moment and on a left-open interval. To memorize the value to assign we introduce for every moment T an auxiliary value $\rho^+(T)$. So to define a run we define simultaneously $\rho(t)$, $\rho^+(t)$, a sequence of TAE $\mathcal{P}(k)$ and some other notions.

By $\mathcal{P}(k)$ will be denoted the TAE obtained after k applications of Time Propagation Rules. We define by induction on k time moments T_k, TAEs $\mathcal{P}(k)$, functions $\rho(t)$ for $t \le T_k$, $\rho^+(t)$ for $t < T_k$ and also:

– for every operator Op we define the set $ExInt[Op, T_k]$ of its intervals of execution of the form $[a, b)$ by describing consecutively their left and right ends a and b situated to the left of T_k,

– for every internal function f of type $\mathcal{X} \to Z$ and $X \in \mathcal{X}$ we define a set $Val[f, X, T_k]$ of values to take as $f(X)$ at some moment greater or equal to T_k,

and the predicate $Tm[f, X, T_k]$ which says that T_k is a moment to change the value of $f(X)$. By default, every element of the set Val is deleted from the set after having been used.

– for each occurrence $Assign$ of an assignment of the form $f(\overline{\eta}) := \theta'$ we define a function $Arg[Assign, T_k]$ that gives X for which this assignment is to change $f(X)$ at T_k or later.

Time Propagation Rules are being applied to each occurrence of T_k in $\mathcal{P}(k)$.

To simplify the recursive definition, extend $\mathcal{T}$ to the left of 0 by, say, some $\varepsilon < 0$, and set $T_0 = 0$, take $\rho(t)$ for $t \in [\varepsilon, 0]$ as defined by the initial state, set $\rho^+(t) = \rho(0)$ for $t \in [\varepsilon, 0)$ and $\mathcal{P}(0) = \{\uparrow(0)P_1, \ldots, \uparrow(0)P_N\}$. All the mentioned sets are empty for T_0 and predicate Tm is false.

Suppose that $T_{k-1} < \infty$, and all the mentioned functions and sets, have been defined for T_{k-1} for some $k > 0$. Let $T_k = MinTm(\mathcal{P}(k-1))$ be the minimum of arrowed occurrences of time in $\mathcal{P}(k-1)$. If $T_{k-1} < T_k$ then first apply (UpDate) and then (Propagation) as described below, otherwise apply directly (Propagation). The procedure (UpDate) is applied when all minimal time occurrences with the same value have been eliminated – in this case the time advances, and we make updates and extend all the values to the next minimal time.

(UpDate): For all internal function $f : \mathcal{X} \to Z$ and for all $X \in \mathcal{X}$ such that $Tm[f, X, T_{k-1}]$ do: if $\#Val[f, X, T_{k-1}] > 1$ then run is undefined for $\tau > T_{k-1}$; if $Val[f, X, T_{k-1}] = \{v\}$ then set $f^{\circ+}(\tau) = v$, otherwise set $f^{\circ+}(\tau) = LeftLim_{t \to T_{k-1}} f^{\circ+}(t)$ for $\tau \in [T_{k-1}, T_k)$. Thus $\rho^+(\tau)$ is extended for $\tau < T_k$. Now set $\rho(\tau) = \rho^+(T_{k-1})$ for $\tau \in (T_{k-1}, T_k]$. After that, each used set $Val[f, X, T_{k-1}]$ becomes empty at T_k, and the others are extended to T_k.

(Propagation): If there is an occurrence of the form $\theta :=\!\!\downarrow\!\!\downarrow (T_k)\theta'$ take it, otherwise take any arrowed occurrence of T_k, and act according to the case:

(A.1) $\uparrow(T_k)\,\theta := \theta'$. Replace this occurrence by $\theta \uparrow\!\uparrow (t') := \theta'$ choosing arbitrary $t' \geq_{delay} T_k$. The point T_k is *the left end of interval of execution* of the assignment under consideration.

(A.2) $\theta \uparrow\!\uparrow (T_k) := \theta'$, where $\theta = f(\eta_1, \ldots, \eta_n)$. Let $X_0 = (\eta_1, \ldots, \eta_n)[\rho(T_k)]$ and $v_0 = \theta'[\rho(T_k)]$. Add v_0 to $Val[f, X_0, T_k]$ and set $Arg[Assign, T_k] = X_0$, where $Assign$ is the occurrence of $\theta := \theta'$ under consideration. Then choose any $t' \geq_{delay} T_k$ and replace (A.2) by $\theta :=\!\!\downarrow\!\!\downarrow (t')\,\theta'$.

(A.3) $\theta :=\!\!\downarrow\!\!\downarrow (T_k)\,\theta'$. Set $Tm[f, X_0, T_k]$ and $Val[f, X_0, T_k] = Val[f, X_0, T_{k-1}]$ for $X_0 = Arg[Assign, T_{k-1}]$. Replace expression (A.3) by $\theta := \theta' \downarrow (t')$ for any $t' \geq_{int} T_k$. The point t' is *the right end of interval of execution* of the assignment under consideration.

(B) Occurrences of $\uparrow(T_k)$ and $\downarrow(T_k)$ different from that of (A). Apply Time Propagation Rules in a evident way. Set $Val[f, X, T_k] = Val[f, X, T_{k-1}]$ for all X and $Arg[Assign, T_k] = Arg[Assign, T_{k-1}]$ for all $Assign$. The intervals of execution of operators are defined also in a straightforward way, say, $\uparrow(t)$**Seq** gives the left end for the operator starting with this **Seq**, and **EndSeq** $\downarrow(t')$ corresponding to this **Seq** gives the right end of the interval of execution.

Each application of the propagation rules as mentioned above give a new TAE which is, by definition, $\mathcal{P}(k)$, and we set $\mathcal{P}(k-1) \vdash \mathcal{P}(k)$.

The induction step of the definition is over.

A run is *total* if it is defined over $\mathcal{T}$ or $\mathcal{T} \cup \{\infty\}$. There are 2 reasons to get non total run, namely, incompatibility of assignments or Zero run, i. e. a run where $T_k \leq const < \infty$ for $k \to \infty$. By induction that follows the definition of run one can prove

Lemma 1 *In a total run the values of internal functions are piecewise constant on intervals of the form* $(t, t']$, $t < t'$.

2.3 On Logic Representation of Runs

Runs of an asynchronous distributed algorithm $\mathcal{A}$ can be described as the set of models of some logical formula. We sketch a first order timed logic over some vocabulary $\mathcal{W}$ to give such a description. Our approach here is straightforward and seems to be less efficient from the point of view of proof search than the logic FOTL in [4] for synchronous block GASM.

The syntax of the logic we consider consists, in a way, of two sublanguages $\mathcal{L}_0$ and $\mathcal{L}_1$. The language $\mathcal{L}_0$ corresponds to the language of $\mathcal{A}$, and $\mathcal{L}_1$ describes how time propagation rules determine runs. Thus, $\mathcal{L}_1$ gives an interaction of syntactic part of the time propagation with the values of arguments and functions. So we are to distinguish, say, term as a word and the value of the term. In particular for this reason, we assume that $\mathcal{A}$ does not use time variables.

Notice that, given an algorithm, we can write down all occurrences of its processes and operators. By Op below we will denote a meta-variable for such occurrences. We could take a logic where Op would be a variable, but such a logic would be more complicated from the point of view of formal verification.

The vocabulary $\mathcal{W}_0$ of $\mathcal{L}_0$ has the same sorts as the vocabulary $\mathcal{V}$ of $\mathcal{A}$, the same constants (static nullary functions) and in place of a function f of type $\mathcal{X} \to Z$ it has a symbol f^0 of type $\mathcal{T} \times \mathcal{X} \to Z$.

The vocabulary $\mathcal{W}_1$ of $\mathcal{L}_1$ is a vocabulary to write down TAEs with time variables in arrow expressions. It contains vocabulary $\mathcal{W}_{TAE}$, words over which, i. e. the set $\mathcal{W}^*_{TAE}$, contain all TAEs. But we need also parameterized TAE. The latter will be represented by terms built with the help of concatenation from constants representing atoms to construct TAEs (key words, delimiters, arrows etc.) and from variables for words and time. Thus, time variables are treated from the point of view of their values when speaking about relations between TAEs (equality, occurrence etc.). Functions and predicates of $\mathcal{W}_1$ permit to describe occurrences of operators, their form, occurrences of arrowed expressions, arguments etc. Any occurrence of a word U in W can be described by the number of its first position in W or as (V_1, UV_2) or (V_1U, V_2) etc., where brackets and comma used here are supposed to be not in the alphabet of W.

We assume that each occurrence of operator in $\mathcal{A}$ has a name, and Op will be used as a variable for such occurrences. Letters Γ, Δ, U, V, W will be variables for words over $\mathcal{W}_{TAE}$, f a meta-variable for internal function which type will be

denoted as $\mathcal{X} \to Z$, X a list of variables of sorts $\mathcal{X}$, v a variable for values of f and T, t, T' ... time variables.

The description of run given above can be written as a formula that defines a unary predicate TAE over $\mathcal{W}^*_{TAE}$, binary predicate $\vdash$, predicates $ExInt(Op, t, [a, b))$, $Val(f, X, t, v)$, $Tm(f, X, t)$ and function $Arg(Assign, t)$ corresponding to sets and function with the same names used in the definition of run (here $Assign$ and Op are meta-variables, thus we have a function $Arg_{Assign}(t)$ for each occurrence $Assign$ of assignment and similar for $ExInt$).

Now run ρ° is a function of time and is represented as a finite set of symbols f°, each f° being of the type $\mathcal{T} \times \mathcal{X} \to Z$ and corresponding to $f : \mathcal{X} \to Z$ of the vocabulary of $\mathcal{A}$. To represent $\rho^{\circ+}$ we use symbols $f^{\circ+}$ of type $\mathcal{T} \times \mathcal{X} \to Z$.

We describe how to write a formula representing runs with comments concerning this description. First, we write a conjunction of formulas related to the initial state.

(RunIni) $\forall\, t \in (\varepsilon, 0]\, IniState(\rho^\circ(t))$, initial values of functions constituting $\rho^\circ(t)$ for $t \in [\varepsilon, 0]$ are determined by the initial state.

(RunPlusIni) $\forall\, t \in [\varepsilon, 0)\ \rho^{\circ+}(t) = \rho^\circ(0)$, here we define the initial values of functions constituting $\rho^{\circ+}(t)$ for $t \in [\varepsilon, 0]$.

(TAEIni) $TAE(\{\uparrow(0)P_1, \ldots, \uparrow(0)P_N\})$ defines the initial TAE.

(ExecLftIni) 0 is the left end of interval of execution of $Prog_j$, $1 \le j \le N$.

(ValueToAssign) For all internal $f : \mathcal{X} \to Z$ and $X \in \mathcal{X}$ predicate $Val(f, X, t, v)$ is $false$ for all $t \in [\varepsilon, 0]$ and for all v of sort of values of f.

This just described formula is connected by conjunction with the formula below describing a recursive step. This formula is a conjunction of 2 formulas – one corresponds to (Update) and the other to (Propagation).

Notations: • $\Phi =_{df} Tm(f, X, T)$. • Predicate $t = MinTm(W)$ means "t is the minimum of times that occur in arrowed expressions of W".

(Update): The prefix $\bigwedge_f \forall T \forall T' \forall \Gamma \forall \Delta \forall X \forall v$ is followed by implication from $\big(\ TAE(\Gamma) \wedge TAE(\Delta) \wedge T = MinTm(\Gamma) \wedge T' = MinTm(\Delta) \wedge T < T' \wedge \Gamma \vdash \Delta \wedge Val(f, X, T, v)\ \big)$ to:

$\forall\, \tau \in [T, T')\ \big[\ \big(\ \Phi \to f^{\circ+}(\tau, X) = v\ \big) \wedge$
$\big(\ \neg\Phi \to \big(\ f^{\circ+}(\tau, X) = LeftLim_{t \to T}\, f^{\circ+}(t, X) \wedge Val(f, X, T', v)\ \big)\ \big)\ \big] \wedge$
$\forall\, \tau \in (T, T']\ \big[\ f^\circ(\tau, X) = f^{\circ+}(T, X)\ \big]$

(Propagation): This formula starts with $\forall T \forall t \forall \Gamma \forall \Delta \forall U \forall V$ and some other universal quantifiers of more technical nature (we do not mention them) and has as its scope an implication. This implication has as its premise $(TAE(\Gamma) \wedge T = MinTm(\Gamma))$ and as its conclusion a conjunction of formulas describing possible situations where one can meet an arrowed occurrence of T. These situations correspond to time propagation rules as in the definition of run. We consider only 3 situations (S1)–(S3).

(S1) $\Gamma = U$ **If** $Guard \Uparrow (T)$ **Then** Op **EndIf** V, where U and V give the global context of the occurrence of $\Uparrow(T)$ under consideration, $Guard$ and Op are a concrete guard and a concrete operator of $\mathcal{A}$. We take the formula $Guard^\circ(T)$ that is constructed from $Guard$ by replacing all terms of the form $f(\overline{\eta})$ by their timed version $f^\circ(T, \overline{\eta}^\circ)$; the operation $^\circ$ goes down to variables and constants

that it does not change. So we continue writing the formula: $Guard^{\circ}(T)$ and $T \leq_{delay} t$ and $\Delta = U$ **If** $Guard$ **Then** $\uparrow(t)Op$ **EndIf** V implies $TAE(\Delta)$ and $\Gamma \vdash \Delta$ and the extension of Val, Tm, Arg to t.

(S2) $\Gamma = U\,\theta \Uparrow (T) := \theta_0\, V$, where $\theta = f(\eta_1, \ldots, \eta_n)$. Let $v_0 = \theta_0^{\circ}(T)$ and $X_0 = (\eta_1, \ldots, \eta_n)^{\circ}(T)$. Here, in addition to the discourse concerning Δ, TAE and $\vdash$, we set $Val(f, X_0, T, v_0)$ and $Arg(Assign, T) = X_0$.

(S3) $\Gamma = U\,\theta :=\!\downarrow\!\downarrow (T)\theta_0\, V$. Here in addition to defining $TAE(\Delta)$ and $\Gamma \vdash \Delta$ we say that for all v such that $Val(f, X_0, T, v)$, where $X_0 = Arg(Assign, T)$ and $Assign$ is the assignment under consideration, there take place $Tm(f, X_0, T)$ and $Val(f, X_0, T, v)$.

3 Refinement of Communication

3.1 Refinements

A refinement of a distributed algorithm is not a simple replacement of an operator by another one which is, in a way, more detailed. Though such a replacement can take place, there may be other changes in the algorithm being refined. We will see it later for Lamport's Bakery. We define refinement semantically, in terms of runs.

Assume that the runs under consideration are total, and the functions of the vocabulary do not have *udef* as their value.

Let $V \subseteq V_1$ be two vocabularies of the type described above. A run over a vocabulary W is an interpretation of sorts and a mapping that give for any $t \in \mathcal{T}$ a value of $f(X^*)$ for each $f \in V : \mathcal{X} \to Z$ for every $X^* \in \mathcal{X}^*$, where $\mathcal{X}^*$ denotes the interpretation of sorts $\mathcal{X}$. Now suppose that an interpretation of sorts is fixed, that means in particular, that every sort in V is interpreted in V_1 as in V.

Projection of a run φ_1 over V_1 onto V (notation: $proj_V(\varphi_1)$) is the run that is the result of deleting from φ_1 all identifiers of $V_1 \setminus V$.

Let $\mathcal{A}$ and $\mathcal{A}_1$ be algorithms respectively over V and V_1.

We wish to compare runs of $\mathcal{A}$ and its refinement $\mathcal{A}_1$ modulo refined operators. As the latter are only in $\mathcal{A}$ but not in $\mathcal{A}_1$ we are to use some abstraction of runs of $\mathcal{A}$ and that of $\mathcal{A}_1$ modulo some sequence of intervals (corresponding to the intervals of execution of operators to refine) supplied with sets of identifiers (corresponding to the identifiers being changed by these operators).

Operation $Weed$ takes as its argument an interpretation φ of vocabulary W (in particular, a run of $\mathcal{A}$ or that of $\mathcal{A}_1$) and a finite set of pairs $\Omega = \{(\alpha_i, F_i)\}_{1 \leq i \leq m}$, where each α_i is a sequence of disjoint time intervals (in increasing order) and each F_i is a finite set of identifiers of W different from input or output ones. The value $Weed(\Omega, \varphi)$ is a mapping φ' from time to values of φ extended with *udef* obtained by the following procedure: for each interval $\alpha_i(j)$ set the values of each $f(X) \in F_i$ equal to *udef* inside α_i.

Consider a set $S = \{Op_1, \ldots, Op_m\}$ of disjoint operators of $\mathcal{A}$. Denote by α_i the sequence of intervals of execution of Op_i and by F_i the set of identifiers changed by Op_i except the output ones. This gives the set Ω.

An algorithm $\mathcal{A}_1$ is a *refinement* of $\mathcal{A}$ with respect S (*operators to refine*) if for every run φ of $\mathcal{A}$ there exists a run φ_1 of $\mathcal{A}_1$ such that $Weed(\Omega, proj_V(\varphi_1)) = Weed(\Omega, \varphi)$.

Algorithm $\mathcal{A}_1$ is a *strong refinement* of $\mathcal{A}$ with respect of a set of disjoint operators of $\mathcal{A}$ (*operators to refine*) if it is a refinement of $\mathcal{A}$ and for every run of $\mathcal{A}_1$ its projection into V is a run of $\mathcal{A}$.

The disjointness of operators in the definition above is not essential as any two (occurrences of) operators are either disjoint or one is a part of the other. We suppose, without loss of generality, that the vocabulary of the refined algorithm intersects the vocabulary of the initial algorithm exactly at inputs/outputs. As a consequence of the fact that $Weed$ do not touch inputs/outputs we have

Theorem 1 *Strongly refined algorithm verifies the requirements iff the non-refined one verifies them.*

3.2 Lamport's Bakery Algorithm

To illustrate how the introduced notions of run and refinement work we consider Lamport's Bakery Algorithm for the Critical Section Problem from [7] deleting all questions of fault-tolerance vaguely discussed by L. Lamport. We rewrite it as asynchronous distributed algorithm, constituted by N programs shown on Fig. 1, where $x_p[1:N]$ is a local array of process p, and the strict order $(a,b) < (c,d)$ means that $a < c$, or $a = c$ and $b < d$. The function $choosing(i)$ (which role is slightly obscure) in original Lamport's Bakery is ensured by the semantics, so it is omitted (as well as in [2]). The idea of the algorithm is as follows. A process wishing to enter critical section (CS) gets a ticket ($number$) calculated in line 3 and then enters Bakery with this ticket. Then he is waiting until all previous tickets were satisfied, and enters CS. Tickets are natural numbers, the number of processes involved is N, CS_p is a predicate indicating that a process p in CS. Instruction CS_p means $CS_p := true$, $\neg\, CS_p$ means that $CS_p := false$. The array x_p serves to get tickets of other processes. How p achieves access to $number_q$ is not discussed and will be the subject of refinement.

L. Lamport does not give rigorous semantics. Our algorithm on Fig. 1 and semantics preserve the main idea of the Lamport's Bakery. To be more precise with respect to Lamport's Bakery we are to permit an arbitrary delay after line 8, however it does not change our reasoning on the whole, so we do not do it. So in our case each process executes $number_p := 0$ infinitely often. Following [7] one can prove, that $BakeryAsyn$ satisfies:
(Safety): $\forall\, pq \forall\, t\ \big(\ p \neq q\ \rightarrow\ \neg\, (CS_p(t)\ \wedge\ CS_q(t)\ \big)$
(No two processes can be in CS at the same time.)
(Liveness):
$\forall\, p \forall\, t\ \big(\ number_p(t) > 0 \rightarrow \exists\, t'\, (t < t' < t + C_1 \cdot \delta_{int} + C_2 \cdot \delta_{ext}\ \wedge\ CS_p(t')\ \big)$
for some natural constants C_1 and C_2.
(Each process that wishes to enter CS eventually enters there.)

Strictly speaking, (Safety) and (Liveness) constitute only part of requirements specification, namely, specification of functioning. There must be present

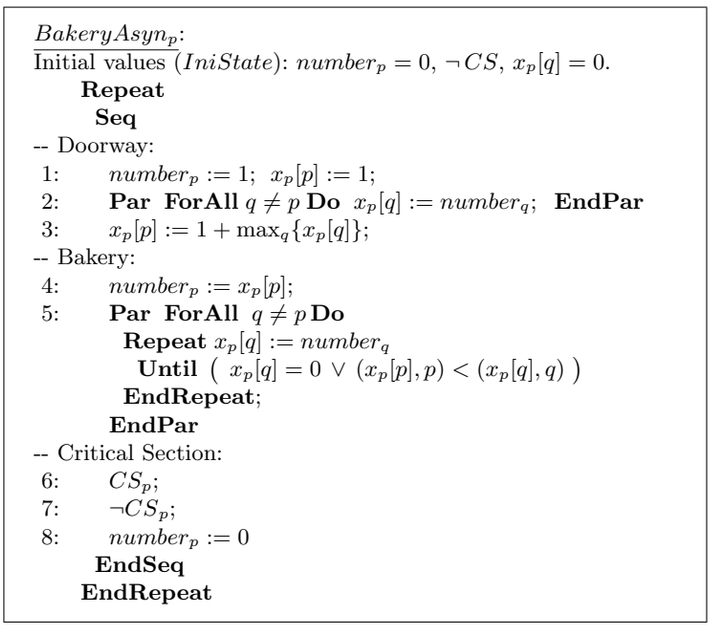


Fig. 1. Lamport's Bakery as Distributed Asynchronous Algorithm.

specification of environment. But in our case, as the algorithm has no outputs, the environment is represented by the semantics of algorithm.

Program for each p, shown on Figure 1, uses its local array $x_p[1 : N]$, and the interaction of the process p with q is represented as the assignment $x_p[q] := number_q$. Our goal is to refine this interaction and describe it in terms of lower level interactions preserving, however, Lamport's "high-level" proof of correctness. To see the problem better we will refer to operator 2 as $Numbers(p, x_p)$ and operator 5 as $Priority(p, x_p)$.

3.3 Refinement of Lamport's Bakery with Parallel Communications

So we wish to refine $Numbers_p$ and $Priority_p$ for all p. In the algorithms described above we may have simultaneous reads or writes. But simultaneous reads or writes implicitly involves solving the problem of critical section, so we are in a vicious circle which we wish to avoid. In Lamport's algorithm there are two types of parallelism, one concerns the interaction between processes, and the other is the internal parallelism of one process. To manage the interaction between the processes we will use a standard communication medium of asynchronous distributed algorithms (based on send/receive messages).

Firstly, we introduce a new sort, namely *Message*. Each message μ is a quadruple (*message#*, *addressee*, *sender*, *contents*), each component being of

the following sort: $message\# \in \mathbb{N}$, $addressee$, $sender \in Process$, $contents \in \mathbb{N} \cup \{?\}$. Symbol ? will mean sending a query on the *number*.

The numbering of messages is necessary only to specify the communication medium as a set – if the messages are not numbered we cannot distinguish two equal messages sent at different moments of time that must be received twice. We assume that a number is attributed automatically and will not mention it.

$Medium$ is a finite set of messages of a priori non bounded size. We assume that every sent message eventually reaches its destination within the external delay. To send messages a process p make assignment to one of its internal functions *send* that implies immediate sending of this message to $Medium$. The messages arriving at p are placed in one or several queues. Extracting a message is done by an assignment of the form $z := FirstQueue$ which implies that the first message becomes the value of z and disappears from the queue. If the queue is empty then the value given by $FirstQueue_p$ is $\perp$. Each queue has its own $FirstQueue$.

A straightforward way of refinement of $BakeryAbstr$ is to replace each $x_p[q] := number_q$ by a sending/receiving of messages preserving the enveloping parallelism. But it does not work, for example in the case of parallel communication, as defined below, we may even have deadlocks or other misfunctioning (see Remark 1 below).

To model parallel communication facility we introduce for each process p functions $send^{\ll}_{p,q}$ (to send its queries), $send^{\gg}_{p,q}$ (to send responses to queries of others), $FirstQueue^{\ll}_{p,q}$ (to receive responses to its queries), $FirstQueue^{\gg}_{p,q}$ (to receive queries of others). In the description of environment one is to include two queues for each p: $Queue^{\ll}_{p,q}$ and $Queue^{\gg}_{p,q}$.

The refinement with parallel communication is on the Fig. 2 and is self-explanatory.

Theorem 2 *Algorithm BakeryRfnP is a strong refinement of BakeryAsyn if its delays are appropriately diminished (that can be easily estimated).*

Proof. We prove two assertions marked below by (A) and (B).
(A) $\forall\, \rho \in Runs(BakeryRfnP) : \ \ proj(\rho) \in Runs(BakeryAsyn)$.

Property (A) is implied by the two following claims which are consequences of the fact that every sent message arrives at its destination within the delay δ_{ext}, and that every query from another process will be responded. In the latter case there may be some internal delay before sending a response, but having been sent, the response will reach its destination within the external delay. We do not estimate concrete delays as this is evident.

Claim 1. *If $[t, t_1)$ is an interval of execution of operator 2 by p in the run ρ then $x^{\circ}_p[q](\tau_q) = number^{\circ}_q(\tau'_q)$ for some $t < \tau'_q < \tau_q < t_1$.*

Proof. Suppose that p has sent a query to q executing *Operator* 2. There is a moment of time such that $Queue^{\gg}_{q,p}$ contains ?. As each process q repeats *Operator* 9.1, we can consider that the value of $w_{q,p}$ from this moment of time is ?. Thus q sends its *number* to p within delay $O(\delta_{int})$.

The Medium will provide $Queue^{\ll}_{p,q}$ with $number_q$ and thus, the assignment $z_{p,q} := FirstQueue^{\ll}_{p,q}$ will give $number_q$ as the value of $z_{p,q}$ and further as the

$BakeryRfnP_p$:
Repeat
Seq
-- Doorway:
1: $number_p := 1;\ \ x_p[p] := 1;$
2: **Par ForAll** $q \neq p$ **Do**
2.1: $send^{\ll}_{p,q} := (q, p, ?);$
2.2: **Repeat** $z_{p,q} := FirstQueue^{\ll}_{p,q}$ **Until** $Contents(z_{p,q}) \in \mathbb{N}$ **EndRepeat**
2.3: $x_p[q] := Contents(z_{p,q});$
EndPar;
3: $x_p[p] := 1 + \max_q\{x_p[q]\};$
-- Bakery:
4: $number_p := x_p[p];$
5: **Par ForAll** $q \neq p$ **Do**
Repeat $send^{\ll}_{p,q} := (q, p, ?);$
Repeat $z_{p,q} := FirstQueue^{\ll}_{p,q}$ **Until** $Contents(z_{p,q}) \in \mathbb{N}$ **EndRepeat**
$x_p[q] := Contents(z_{p,q});$
Until $\big(\ x_p[q] = 0 \vee (x_p[p], p) < (x_p[q], q)\ \big)$
EndRepeat;
EndPar
6–7: $CS_p;\ \ \neg CS_p;$
8: $number_p := 0$
EndSeq
EndRepeat
$\|$
9: **Par ForAll** $q \neq p$ **Do**
Repeat
9.1: $w_{p,q} := FirstQueue^{\gg}_{p,q};$
9.2: **If** $Contents(w_{p,q}) =?$ **Then** $send^{\gg}_{p,q} := (q, p, number_p)$ **EndIf**
EndRepeat
EndPar

Fig. 2. Lamport's Bakery with Parallel Communication.

value of $x_p[q]$. It is clear that to send/receive a message the algorithm spends at most $2\delta_{ext}$ time plus some more delays δ_{int}. Remark that each $Queue_{q,p}$ has at most one element at a given moment. ■

Claim 2. *If $[t, t_1)$ is an interval of execution of operator 5 by p in the run ρ then for all $q \neq p$ we have $x_p[q](\tau_q) = number_q(\tau'_q)$ for some $t < \tau'_q < \tau_q < t_1$ and $\big(\ x_p[q](\tau_q) = 0 \vee (x_p[p](t'), p) < (x_p[q](\tau_q), q)\ \big)$ for some $t' < t$.*

Proof. As in the previous proof, using the properties of the Medium, one can easily deduce Claim 2. ■

(B) $\forall \rho_0 \in Runs(BakeryAsyn) \exists \rho \in Runs(BakeryRfnP)$:
$Weed(proj(\rho)) = Weed(\rho_0)$.

Let $\rho_0 \in Runs(BakeryAsyn)$. We construct ρ by induction on the changes of the values of ρ_0. For the moment 0 the initial values of the both considered algorithms are compatible. Clearly the only cases to consider are the executions of operator 2 and operator 5 of the $BakeryAsyn$ algorithm which are refined by $BakeryRfnP$. Remark that identifiers of operator 2 and operator 5 of different processes are different, and intervals of execution of operator 2 and that of operator 5 for the same p are disjoint.

We take an interval $\zeta = [t, t_1)$ of execution of operator 2. And within this interval of time, we construct an appropriate execution of $BakeryRfnP$. Take all the moments τ_q and τ'_q in ζ such that $x_p^\circ[q](\tau_q) = number_q^\circ(\tau'_q)$ for $BakeryAsyn$.

Firstly, we choose delays such that for each q, *Operator* 2.1 ends before τ'_q, say at t_q in $BakeryRfnP_q$ that is $Queue^{\gg}_{q,p}$ contains the query of p at time t_q. Then we can choose delays such that q executes *Operator* 9.2 and gives $(p, q, number_q^\circ(\tau'_q))$ as the value of $send^{\gg}_{q,p}$. Then delays are chosen so that p executes *Operator* 2.2 and 2.3 and finally gives $number_q^\circ(\tau'_q)$ as the value of $x_p[q]$ at time τ_q. The value of the array x_p at the end of ζ is the same in both executions of the two algorithms.

We can deal in the same way with operator 5. ∎

Theorems 2 and 1 imply

Corollary 1 *The refined algorithm BakeryRfnP satisfies (Safety) and (Liveness).*

Remark 1. The well functioning of Lamport's algorithm leans particularly upon 'good' communications between processes. If we wish to make precise such exchange of information, the very first idea is to focus only on the concerned operators that is operator 2 and operator 5 in which communications occur. However, this idea does not lead to a correct refinement. Suppose we authorize only communications during the intervals of time when these two operators are running. As a consequence, a process can only receive a value 1 or greater as a response to its query since during the execution of these two operators $number_r \neq 0$ for any process r.

Hence, a run of Lamport's Bakery, where each process stays idle from time 0 with exception of one who reaches critical section, cannot be a run of our 'refined' algorithm.

Let p, q be two processes and let us consider the following run of Lamport's Bakery:

- $number_q(t) = 0$ for $t \in [0, \infty)$
- $\neg CS_q(t)$ for $t \in [0, \infty)$
- $number_p(t) = 0$ for $t \in [0, 1)$
- $number_p(t) = 1$ for $t \in [1, 2)$
- $number_p(t) = 2$ for $t \in [2, 4)$
- $\neg CS_p(t)$ for $t \in [0, 3)$
- $CS_p(t)$ for $t \in [3, 4)$
- $number_p(t) = 0$ for $t \in [4, \infty)$
- $\neg CS_p(t)$ for $t \in [4, \infty)$

where the intervals of execution of operators 2 and 5 for process p are respectively $[1, 2)$ and, say, $[2.5, 3)$.

An execution of the 'refined' algorithm we consider is supposed to have the same values for $number_p$, $number_q$, CS_p and CS_q except during these two intervals of time. Therefore we must have $number_q(t) = 0$ and $\neg CS_q(t)$ for $t \in [0, 1) \cup [2, 2.5) \cup [3, \infty)$. To answer to the query of p during $[1, 2)$, q must be running operator 2 or operator 5. So q will execute operator 2 during $[1, 2)$. In order to have its number back to value 0, q must execute operators $3 - -8$ before time 2 : it is not possible because during that interval of time, $number_p = 1$ and the loop in operator 5 cannot end.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. Technical Report 29, DEC Systems Research Center, Palo Alto, California, August, 14 1988.
2. E. Börger, Y. Gurevich, and D. Rozenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.
3. M. Broy. Compositional refinement of interactive systems. *J. of the Assoc. Comput. Mach*, 44(6):850–891, 1997.
4. D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: Basic properties and a decidable class. 37 pages, 1999. To appear in *J. of Pure and Applied Logic*.
5. G. Goos, A. Heberle, W. Loewe, and W. Zimmermann. On modular definitions and implementations of programming languages. In *Proc. of the Intern. Workshop on Abstract State Machines (ASM'2000), March 20–24, 2000, Switzerland, Monte Veritá, Ticino*, pages 174–208. ETH, Zürich, 2000.
6. Y. Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–93. Oxford University Press, 1995.
7. L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of ACM*, 17(8):453–455, 1974.
8. J. Ostroff. Composition and refinement of discrete real-time systems. *ACM Trans. on Software Engineering and Methodology*, 8(1):1–48, 1999.
9. M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science. Vol. B: Formal Models and Sematics*, pages 677–788. Elsevier Science Publishers B.V., 1990.

Objects + Views = Components?

Martin Odersky

École Polytechnique Fédérale de Lausanne

Abstract. In support of flexible composition, components need to be adaptable and they need to have first-class plugs. Abstract state machines and object-oriented programming each satisfy one of these requirements very well but satisfy the other only incompletely. This paper describes views, a language construct which provides for both requirements in a balanced way.

1 Introduction

Increasingly, software is developed from pre-existing components rather than being built from scratch. Component middleware such as Corba [25], COM [2] or Java-Beans [6] allow applications to be constructed with the help of binary-components and foster the development of a component market [26]. Components can also take the form of services which are implemented as separate software systems accessed over the internet [30].

Looking beyond the technical details, components are parts which exist to be composed. The most important difference with respect to modules in traditional top-down development is that future compositions of a component are unknown to the component's authors. To be useful, a component library therefore needs to provide high flexibility in component reuse and composition. It has been argued that this is a difficult task, in particular because the object-oriented languages used to implement components do not provide the right abstractions [10].

This paper explores language design issues in the area of composition. We argue that a component is composable only if two requirements are fulfilled: The component needs to be *adaptable* and its plugs should be *first-class values*.

A component is *adaptable* if clients can enrich its interface after the component has been constructed and delivered. Such changes consist typically in adding some data fields or methods which the original component did not yet support. The methods are to be defined in terms of the component's existing public interface. Such additions should not affect a component's source code. After all, source code changes by clients are impossible for binary and service components and pose severe version control problems for components existing in source form.

The need for adaptations arises because at the time when a component is designed and implemented one often does not know precisely in which contexts the

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 50–68, 2000.

component will be used. As a simple example, consider the task of writing a class for lists. Which methods should this class offer? A fairly small implementation could be:

```
class List[a] = {
  def head: a = ...
  def tail: List[a] = ...
  def isEmpty: Boolean = ...
}
```

Of course there are many more useful functions to be included in the List class. Functions to select elements inside the lists, to form sublists, to concatenate lists, to filter elements according to some criterion, or to map functions over all elements come to mind. Furthermore, one might want to treat lists as representations of sets, in which case membership, union, intersection and set difference also should be supported. And one can think of many other useful operations on lists. The hard question is where to stop. Meyer [20] recommends a "shopping list approach" to class design where essentially all useful methods one can think of should be added to an abstraction. In the case of lists, this would probably lead to many hundreds of methods, and even then clients of the list abstraction will likely miss a method they require. It seems preferable to provide a more compact list definition and let clients customize this definition as they require. The criterion for a complete class design would then be that every useful method for the class *can* be defined in terms of the public interface, not that the method already *is* defined. The "can"-completeness criterion is much easier to fulfill; for instance our simple implementation of lists above is already complete.

```
val SymTab {
  type Scope
  type Symbol = {
    def name: String
    def tp: Type
    def location: Scope
    ...
  }
  def lookup (scope: Scope, name: String): Symbol = ...
  def enter (scope: Scope, sym: Symbol): Boolean = ...
  def newScope (outer: Scope): Scope = ...
}
```

Fig. 1. Symbol Table Module

As another example, consider the task of constructing a component for symbol tables in a compiler. Specifications and algorithms for symbol tables are potentially portable over many compiler implementations. Figure 1 shows a sketch of

a generic implementation. However, it's not clear what attributes should go into a symbol, as is illustrated by the ... in the Symbol type. Clearly, symbols will have a "name" attribute. One might also settle on fields that contain a symbol's type and location. But there are also other potential attributes which depend on the situation in which a symbol table is used. For instance, if our compiler has a code generator, symbols will need a field in which to store their address. On the other hand, if the compiler is coupled with a source code browser, we might need an additional attribute in a symbol which contains the list of all usage points of that symbol. Again, the precise definition of some aspect of the symbol table component depends on the contexts the component is used.

In classical structured programming, where data structures and code are separated, symbol tables would be adapted by changing their code, filling in the ... in the Symbol type as required by the application. But source code adaptation is impossible or at least undesirable in a component setting. Furthermore, the necessary adaptations to the symbol table module also violate the encapsulation principle, since fields needed only by the code generator module are placed in the common Symbol data structure which is accessible to all. Hence, better solutions for adaptation are called for.

The other requirement for components has to do with how flexibly they can be composed. One usually regards the run-time embodiments of a component's interfaces as its "plugs". In object-oriented languages plugs are often realized as objects which have methods which are accessible to clients. A plug is *first-class* if it can be treated like a normal value. In particular, one should be able to pass plugs as parameters to functions, and it should be possible to construct data structures with plugs as elements. As a very simple example for this kind of flexibility, consider the task of displaying the information associated with a symbol. Since there are so many different devices on which this information could be displayed, it makes sense to split the task in two. Symbols only provide a method toString, which converts the information associated with the symbol to a string. In other words, symbols support the interface

```
type Printable = {
  def toString: String
}
```

Then, a display device could provide a general display service for objects that "know" how to turn their information into strings:

```
def print(obj: Printable) = ...
```

If dev was such a device and sym was a symbol it would then be possible to write dev.print(sym). Of course, this assumes that symbols are values that can be passed to the print function. In particular, the type Symbol must be compatible with the type Printable.

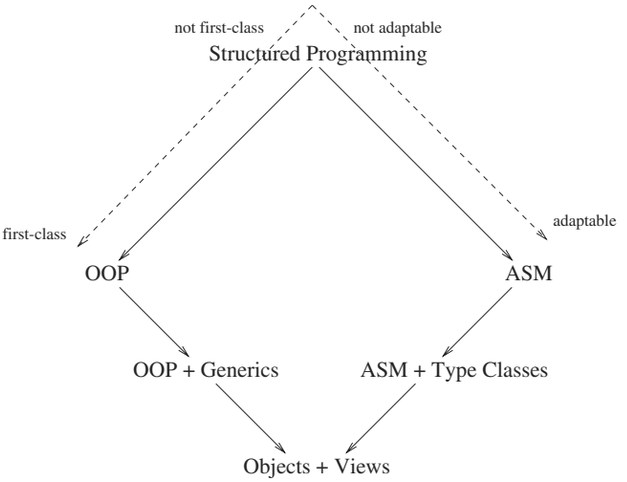


Fig. 2. Support for adaptability and first-class plugs.

Since adaptability and first-class plugs are natural requirements for software components, one might expect that mainstream languages would provide support for both. Maybe surprisingly, this has not been the case so far. Figure 2 summarizes the state of the art, which will be further discussed in the next sections. Even though structured programming languages such as Ada or Modula-2 provide support for modules, which can be seen as an embodiment of components, the components thus defined are neither adaptable, nor do they have first-class plugs. Object-oriented programming leads to components with first-class plugs, but these components are not adaptable. Conversely, the approach taken by abstract state machines lends itself to the construction of components which are adaptable, but these components do not have first-class plugs. More advanced type systems and design patterns can in each case provide some of the missing functionality. Refining object-oriented programming with bounded genericity can provide adaptability to some degree, whereas refining functional programming and ASM's with a construct such as Haskell's type classes can provide some of the benefits of first-class plugs.

This paper presents *views* as a simple technique to provide both adaptability and first-class plugs. The technique uses standard object-oriented modeling techniques to model first-class plugs. A view is a syntactic construct which provides new methods for objects of an existing class, thus providing implementations for additional object interfaces. Views that extend a component can be distributed over arbitrary client components.

The discussion in this paper is in the context of statically typed languages. As notation for our examples, we use Funnel [24], together with some hypothetical extensions.

The rest of this paper is structured as follows. Section 2 discusses strengths and shortcomings of the abstract state machine approach to software composition. Section 3 discusses the same for the object-oriented approach. Section 4 presents views. Section 5 discusses related work and Section 6 concludes.

2 Abstract State Machines

The abstract state machine approach reverses the usual relationship between functions and data. Rather than having mutable data structures which are operated on by immutable functions, we have immutable data, which are operated on by both mutable and immutable functions. This makes use of the following analogy between function application and selection.

$$f(x) \quad \equiv \quad x.f$$

In other words, a data selector can be seen as a single argument function over the selected data domain and *vice versa*. This equivalence makes sense on the left-hand sides of assignments as well:

$$f(x) := E \quad \equiv \quad x.f := E$$

We can thus re-interpret assignment of an expression to a variable field f of a record x as an update of a mutable function f at the position x. If we apply this re-interpretation consistently, we arrive at only immutable data, which serve as index structures for possibly mutable functions. The immutable data structures can be described as algebras, which leads to a simple formal treatment of state change. This treatment lends itself to the verification of properties of imperative programs with pointers, which are very hard to prove otherwise [23].

Abstract state machines add to this the idea of specifying change by a global set of rules, which collectively specify an execution step. The question whether execution is described by a global set of rules or by more conventional statements with control flow constructs is irrelevant to the discussion in this paper and will henceforth be ignored. In any case, the abstract state machine approach leads very naturally to components which are adaptable.

For instance, consider the question which functions to provide together with a List type. Abstract state machine programmers would do the same functional programmers do – they would define List as an algebraic data type which can then be accessed by pattern matching. Using Funnel as a notation, the algebraic data type is expressed as a **class** with two **case**s and the pattern match is expressed by a call to the match method of a list. For instance, here is the definition of a list together with two representative functions over lists:

```
class List[a] = {
  case Nil
  case Cons (head: a, tail: List[a])
}

def append (xs: List[a], ys: List[a]) = xs.match {
  case Nil: ys
  case Cons (x, xs1): List.Cons (x, append (xs1, ys))
}
def map [a,b] (f: a → b, xs: List [a]) = xs.match {
  case Nil: xs
  case Cons (x, xs1): List.Cons (f (x), map (f, xs1))
}
```

All operations over lists can be expressed in terms of pattern matching. Hence, a list operation can be placed in an arbitrary module, and new list operations can be added in clients of the list abstraction as needed. In a word, lists designed as algebraic data types are adaptable. This might be too trivial to mention except for the fact that lists represented as objects which export all their methods are not adaptable in the same way, as clients cannot add new methods to an existing list. This point will be further discussed in the next section.

The abstract state machine approach also ensures adaptability for mutable functions. For instance, consider the situation where a symbol table module as given in Figure 1 is adapted to work with a code generator client. The code generator needs to maintain with each symbol its address. This could be achieved by defining a mutable function from symbols to their addresses in the code generator module:

```
var adr (sym: Symbol): Int := 0
```

The := 0 part in the declaration above defines the initial value of the mutable function to be 0 over all of its domain. The code generator module could then enter a symbol's address by an indexed assignment such as

```
adr (sym) := nextLocalAdr
```

and it could access a symbol's address by calling adr(sym). No change to the symbol table module is necessary. Furthermore, since addresses are dealt with only in the code generator, the definition of adr can be kept local to the code generator module, so that we have gained encapsulation as well as adaptability.

This is all good news. However, the abstract state machine approach does not satisfy our second criterion, that components have first-class plugs. For instance, we still cannot define a general print service with a method print(p: Printable) which can be applied to symbols. Indeed, neither classical structured programming nor abstract state machines provide a subtype relation which would let us

treat a Symbol as a special instance of the more general Printable abstraction. This is not an oversight either. To be useful, the Printable abstraction must define a method such as toString, which turns a Printable into a String. Hence, data must be an abstraction which includes functions operating on the data. This view is at odds with the data structuring approaches of functional programming and abstract state machines, which treat data as domain structures for functions defined elsewhere. In a sense our two criteria of adaptability and first-class plugs seem to be at odds with each other. First-class plugs require grouping functions with data whereas adaptability requires grouping functions separately.

The functional programming language Haskell [11] pioneered the concept of *type classes* [29], which can to some degree make up for the lack of first-class plugs in functional systems. It would be quite possible to extend this construct to abstract state machine languages as well.

A type class represents a shared property of a set of types – alternatively, it can be identified with the set of types which satisfy the property. A property states that a type supports a given set of functions with given signatures. For instance, here is a declaration of a type class[1].

```
type class Printable where {
  toString :: a → String
}
```

This states that a type T belongs to the type class Printable if there is a function toString, which takes a value of type T and yields a value of type String. Types become members of type classes by explicit **instance** declarations. Example:

```
instance Printable Symbol where {
  toString (sym) = ...
}
```

The **instance** declaration above declares Symbol to be an instance of type class Printable and provides the implementation of the toString method of Printable.

One can now define functions which are polymorphic over all types which are instances of a given class. Example:

```
print :: Printable a ⇒ a → ()
print x = ... toString (x) ...
```

This defines a function print which takes as parameters values of type a, where a is an arbitrary instance of type class Printable. The call to toString in print will invoke the method which is appropriate for the concrete type to which the type variable a is instantiated. For example, if sym is a Symbol then the type variable a is instantiated to Symbol and print (sym) invokes the toString method given in

[1] We use Haskell syntax except for writing **type class** where Haskell uses **class**.

Symbol's instance declaration. The type Printable a ⇒ a → () is called a *qualified type* [12] and the (Printable ⇒) prefix is called a *context*.

Representing component interfaces as instances of type classes, one can hence write generic functions which work with any component plugs which are instances of a given type class. This yields many of the benefits of first-class plugs, but some shortcomings remain. First, generic pluggability is achieved in a somewhat roundabout fashion. Second, we still cannot define data structures over component plugs. Consider for instance a data structure of lists whose elements are printable components, not necessarily all of the same type. One might try to express this as a List[Printable], but this would be ill-formed, since Printable is a type class, not a type. The qualified type Printable a ⇒ List[a] would not do either, since that type describes a list of objects which all have the same instance type of Printable. One could go further, expressing heterogeneous data structures by means of existential types [18], but this comes at the price of even greater conceptual complexity.

3 Object-Orientation

Instead of starting with an adaptable architecture and trying to emulate first-class plugs, mainstream programming has instead favored the object-oriented approach, which has first-class plugs built in, yet needs additional provisions to achieve some degree of adaptability.

In object-oriented design and programming, one groups data (called *instance variables*) and functions operating on the data (called *methods*) together in an object. Data and functions collectively are called the *fields* of an object. There is a subtyping relation (written <:) between object types, which lets one use an object with a certain set of fields in a context where only a subset of the fields is required. This provides all we need for first-class plugs. For instance, to make symbols printable, it suffices to make the type Symbol a subtype of Printable, which requires the definition of a toString method in Symbol:

```
class SymTab = {
  class Symbol <: Printable = {
    ... (fields as before) ...
    def toString: String = name ++ ":" ++ tp.toString
  }
}
```

Then we can write:

```
val sym = new SymTab.Symbol
...
dev.print (sym)
```

Are objects also adaptable? One might think so, because with *inheritance* it is possible to define a subclass which adds new fields to an existing class. For instance, it is possible to define in the code generator class CodeGen a new class Symbol which inherits from the original symbol class as defined in module SymTab:

```
class CodeGen = {
  class Symbol <: SymTab.Symbol = {
    var adr: Int
  }
  ...
}
```

Instance objects of class CodeGen.Symbol have all the fields of original symbols, plus the mutable adr field. So it seems we have managed to adapt symbol table modules to a code generator client. But closer inspection shows us that this is not really the case. There are two problems with adaptation through inheritance. One concerns types, the other object creation. Looking at types first, we notice that function lookup in our symbol table module still returns objects of type SymTab.Symbol, not objects of type CodeGen.Symbol. So the added adr information does not show up in the type.

```
class Symtab {
  ...
  class Symbol ...
  def lookup (scope: Scope, name: String): Symbol
  ...
}
```

To access the adr field of a code generator symbol stored in the symbol table, we need a dynamic type cast. In other words, we can gain adaptability only by subverting the type system, and running the risk of dynamic type errors. A safer alternative makes use of type parameterization. For instance, here is a re-formulation of the symbol table class using *bounded genericity*:

```
class Symbol <: Printable = {
  ... (fields as before) ...
}
class SymTab [s <: Symbol] = {
  def lookup (scope: Scope, name: String): s = ...
  def enter (scope: Scope, sym: s): Boolean = ...
  def newScope (outer: Scope): Scope = ...
}
```

The class for symbol tables is now parameterized by a type variable s which represents the actual type of symbols stored in the table. The only requirement

on instantiations of s is that they are subtypes of Symbol. Some gluing is needed at the top-level of an application to instantiate the class to the right symbol type:

```
val symTab = new SymTab[CodeGen.Symbol]
```

The other problem with achieving adaptation with inheritance has to do with object creation. Taking our symbol table example, we note that we have not looked yet at code which creates symbols to be stored in the table. Let's say symbols are created during tree attribution, in class Attr:

```
class Attr = {
  ...
    new Symbol(name, tp)
  ...
}
```

This is not yet precise, because we have not yet specified which symbol should be created during tree attribution. If it is the original SymTab.Symbol that is created, we are missing an adr if we want to couple tree attribution with code generation. On the other hand, if it is a CodeGen.Symbol which is created, we have hardwired the relationship between tree attribution and code generation. Now it would be the tree attribution part which lacks adaptability, because it would not be able to deal with different kind of symbols, for instance those required by a source code browser. To overcome this problem, we can make use of the *Factory* design pattern [7]. Factory objects would have an interface of the form:

```
type Factory [t] = {
  def make (...): t
}
```

There would have to be an appropriate factory for every subtype of symbol which is being defined. E.g. in class CodeGen:

```
class CodeGen = {
  ...
  val SymFactory: Factory[Symbol] = {
    def make(name: String, tp: Type): Symbol =
      new CodeGen.Symbol(name, tp)
  }
  ...
}
```

The Attr class will be parameterized with the factory which is to be used for symbol creation. We also have to parameterize Attr with the actual type of symbols to be created. This yields:

```
class Attr [s <: Symbol] (symFactory: Factory [s]) = {
  ...
  symFactory.make (name, tp)
  ...
}
```

Even more gluing is needed at top-level, to obtain the correct instantiation of the tree attribution module:

```
val symTab = new SymTab [CodeGen.Symbol]
val attr = new Attr [CodeGen.Symbol] (CodeGen.SymFactory)
```

The preceding discussion has shown that it is possible to obtain a certain degree of adaptability in object-oriented designs, but this requires planning. If we had not foreseen the possible later extension of the symbol type, we would not have provided the necessary parameters for subtypes and factories. The hooks necessary for enabling future extensions can clutter programs considerably, in particular if we envisage multiple coexisting extensions. For example, the current compiler framework would enable a single extension of Symbol, say for a code generator or a browser, but it would not yet enable multiple coexisting extensions, say where a symbol has both an adr field for a code generator and a uses field for a browser. Multiple extensions can be supported by stacking extensions on top of each other but this requires even more complex protocols.

4 Views

In the previous two sections, we have discussed two program structuring methodologies which have complementary strengths. Each methodology supports one of our two criteria of adaptability and first-class plugs, but supports the other only incompletely and at the price of complex language features and design patterns. In this section, we propose a more symmetric combination which can address both criteria equally well. This combination arises by translating concepts first developed for type classes into an object-oriented setting.

When comparing type classes in functional programming and object-oriented type systems, we find some strong analogies:

type class	$\approx$	type
instance relation	$\approx$	subtype relation
instance declaration	$\approx$	subtyping clause

A type class corresponds to a type in the object-oriented setting. For instance, instead of having the type class Printable, we would have the type Printable, which has all "printable" types as subtypes. Consequently the instance relation between types and type classes becomes the subtype relation between types.

Finally, an instance declaration in Haskell corresponds to a subtyping clause of a class declaration. The analogy highlights an important difference between the approach of Haskell and the one of object-oriented languages: in the latter, a subtyping clause is textually part of the extending class whereas in Haskell, an instance declaration can be given anywhere. Haskell's approach is adaptable, in that instance declarations for a type can be given after the type is defined. The object-oriented approach is not, since adding new subtype edges requires changes in the source code of the subtype.

Languages with structural subtyping do not need explicit subtyping clauses in class declarations. Instead, the subtype relation is determined only by the structure of the classes: Classes with more fields become subclasses of classes with fewer fields, as long as the types of common fields are the same or stand in turn in a subtyping relation. This gives some more flexibility in the subclass relation. But it still does not solve the adaptation problem since it does not provide a way to augment a given class with new fields.

Seeing these disparities it seems promising to provide a construct on the object-oriented side which matches more closely the distributed nature of instance declarations. This construct is called a *view*. A view adds new fields to values of an existing type. For instance, the following view declaration would add the method toString to all values of type Symbol.

```
view (this: Symbol): Printable {
  def toString: String = this.name.toString ++ ":" ++ this.tp.toString
}
```

A view takes the form of an unnamed function, which takes a single parameter of some type A and yields an object of type B. The view establishes a subtype relationship between A and B by giving declarations of all fields of B which are not yet present in A. All fields of A are inherited; i.e. they form implicitly part of the resulting object. It is only possible to add new fields to a class with a **view**, not to change existing fields.

Subtype Clauses

Subtype clauses can be regarded as syntactic sugar for view declarations with empty bodies. For instance, our previous class declaration

```
class Symbol <: Printable = {
  ...
  def toString: String = name ++ ":" ++ tp.toString
}
```

would be equivalent to

```
class Symbol = {
  ...
  def toString: String = name ++ ":" ++ tp.toString
}
view (this: Symbol): Printable = {}
```

Unlike a subtyping clause, a view declaration can be given anywhere in the program; it need not appear textually adjacent to the declaration of the view's parameter class.

Variables in Views

Besides methods, views can also define mutable variables. Here is an example:

```
type Adr = { var adr: Int }
view (sym: SymTab.Symbol): Adr = {
  var adr: Int
}
```

These declarations, when placed in the code generator module of a compiler, would be equivalent to the mutable function which we have used in the abstract state machine structuring:

```
var adr(sym: Symbol): Int
```

The only difference concerns selection syntax – we use the object-oriented version sym.adr instead of the previous adr(sym).

Encapsulation

More interestingly, we can also have information encapsulated in views. Consider for instance an encapsulated abstraction for address fields, which lets only even addresses be stored. This is accomplished as follows.

```
type Adr = {
  def setAdr (x: Int): ()
  def getAdr: Int
}
view (sym: SymTab.Symbol): Adr = {
  var adr: Int
  def setAdr (x: Int) =
    if (x % 2 == 0) adr := x
    else error ("bad address")
  def getAdr = adr
}
```

This view defines a variable adr along with setter and getter methods. Since the variable is not part of the view's target type Adr, it is not accessible to clients of the view.

Conditional Views

Sometimes, a parameterized type can implement a view only if some constraints on the type parameters are satisfied. Consider for instance structural equality as implemented by the interface

```
type Eq [a] = {
  def equals (other: a): Boolean
}
```

A type T supports structural equality testing if it implements the unary equals method which takes other values of type T as parameters. In other words, T supports structural equality testing if T <: Eq[T]. Now, consider the parameterized type of lists, List[a]. Should lists support structural equality testing? Clearly, we can compare lists structurally iff we can compare the list elements. This can be expressed by the following, *conditional* view declaration.

```
view [a <: Eq [a]] (this: List[a]): Eq [List [a]] = {
  def equals (that: List [a]) = {
    this.isEmpty && that.isEmpty ||
    !this.isEmpty && !that.nonEmpty &&
    this.head.equals (that.head) && this.tail.equals (that.tail)
  }
}
```

The [a <: Eq [a]] construct introduces a type parameter a for the view which is bounded by Eq[a]. The type variable can hence be instantiated to any type T which is a subtype of the bound Eq[T]. Note that this gives us a parameterized type which has or lacks method equals, depending on whether or not the type of the listed elements has the same method.

Semantics

The rest of this section discusses detailed design and implementation choices for views. The most fundamental design choice concerns name resolution of fields defined by views. Such resolution can be static, based on compile-time types or it can be dynamic, based on run-time types. To see the difference, consider a variant of our symbol table code. Assume that we want to further refine the type of symbols into variable symbols and function symbols:

```
type Symbol = {
  ... (fields as before) ...
}
type VarSymbol <: Symbol = { ... }
type FunSymbol <: Symbol = { ... }
```

In the code generator, one might need a function loadAdr which returns the instruction for loading a given symbol's address. The result of this function will depend on the kind of symbol encountered. It is therefore reasonable to implement loadAdr as a method of type Symbol, to be overridden in subtypes. But as before, we want to keep the source code in Symtab unchanged. Again as before, we resort to views for implementing the new functionality in the code generator client. Instead of a single view we now need three views, one for each kind of symbol:

```
type Loadable = {
  def load: Code
}
view (sym: Symbol): Loadable = {
  def loadAdr: Code = error ("can't load")
}
view (sym: VarSymbol): Loadable = {
  def loadAdr: Code = LLA (sym.adr)
}
view (sym: FunSymbol): Loadable = {
  def loadAdr: Code = LCA (sym.adr)
}
```

Can this work? Assume that the code generator calls the loadAdr method of a symbol it retrieved (via lookup) from the symbol table. The static return type of lookup is Symbol. So if we perform name resolution based on the static type, the loadAdr will always result in error("can't load"). In an object-oriented language such behavior would be counter-intuitive. We therefore pick dynamic name resolution, where the loadAdr method is chosen according to the run-time type of the symbol returned from lookup. If the returned symbol represents a variable, we invoke the loadAdr method of the view for VarSymbol. If the symbol represents a function, we invoke the loadAdr method of the view for FunSymbol instead. Only if the returned symbol is neither a variable nor a function, the loadAdr method of the view for the base class Symbol is invoked.

In the last example the call to loadAdr was resolved as if loadAdr was a regular method in Symbol, which was overridden in VarSymbol and FunSymbol. We can indeed often view a program with views as equivalent to a program with multiple inheritance where all views are inlined in their base classes. View methods then

become regular methods and target types of views become elements of subtyping clauses. The correspondence with multiple inheritance programs serves as a guide in the definition of the semantics of views and the context-dependent syntax rules governing them.

The context-dependent syntax rules are based on the concept of the *view graph*. The view graph VG of a project has as nodes the set of types defined in all source files of the project and as edges the set of views between types. Subtyping clauses are also interpreted as view edges. Furthermore, we regard the methods defined in a class as defined in an additional view which goes from a new node representing a class implementation to the proper class type itself. A field which occurs textually in a record type T is said to be *defined* by T. A field which occurs textually in a view V is said to be *implemented* by V. The subtyping ordering $\leq$ on classes induces an ordering on views as follows: A view from S to T precedes a view from S' to T' if $S \leq S'$. We then require the following:

- The view graph must be acyclic.
- A view from S to T may only implement fields which are not already defined in S.
- When selecting a field f from an expression e of type T, there must be exactly one type reachable from T in VG which defines f.
- Let I be the node representing a class implementation and let T be a type reachable from I in VG. Then every field f in T must have a best implementation for I. This means: First, there must be a view on a path between I and T which implements f. Second, among all the views on paths between I and T which implement f, there must be one view which precedes all others according to the induced subtyping ordering on views. This requirement corresponds to the usual requirement for multiple inheritance that inheritance of fields must be unambiguous.

It is worth noting that some of these rules are global in that they require knowledge of the complete view graph. To check such global requirements in a system with separate compilation, one could delay some type checking until link time. As an alternative one could also maintain a project-wide repository in which all current view declarations are maintained. Consistency of the view graph can then be checked incrementally, as views are added and deleted.

5 Related Work

The shortcomings of the object-oriented approach in the area of adaptability have been known for some time. They have given rise to a varied body of research. The work on subject-oriented programming [9] identified the need for multiple, independent roles played by a single object. Aspect-oriented programming [17], hyperspaces [27], and adaptive programming [21] all address this requirement with systems which act as source to source transformations. The idea is in each

case that different aspects of a program are defined in separate source files which are then merged using a "weaver" tool. Binary component adaptation [16] provides mechanisms to perform similar changes on Java class files. Both weavers and binary component adaptation are possible approaches for implementing views. Compared to these approaches, views are more restrictive in that they support only strict extensions of existing classes. One advantage of this restriction is that strict extensions commute, so that the semantics of the weaver tool becomes a non-issue.

Multi-methods [1,4] are another approach to extend the limitations of pure object-oriented programming. Here, a method is no longer attached to a single class, and method selection takes the dynamic type of all method arguments into account. Like views, multi-methods can be defined independently of their argument classes. But multi-methods are more general than views in that they abandon the concept of a distinguished receiver argument and in that they require multiple dispatch. Open objects [19] are an idiom of multi-methods which is comparable to views.

Like views, Haskell's type classes allow a distributed definition of functionality over data types. Qualification of type variables in a qualified type corresponds to F-bounded qualification of views [3]. A number of different designs for type classes have been developed [8,22,12,13,14]. Our view proposal is most closely related to *parametric type classes* [5], in that the bound of a type variable `a` may have additional parameters other than `a` itself. Type classes are usually studied in systems without subtyping. Where subtyping is added [15], name resolution is based on static types, whereas views are based on dynamic types.

A language construct called "view" has also been proposed for functional programming [28]. These views serve as alternative decompositions of sum types such as Haskell's algebraic data types. By contrast, the views presented here provide alternative accesses to product types such as records or objects. Both designs have in common that a view can be defined in program parts other than the one which defined its base type.

6 Conclusion

Do objects plus views equal components? This paper has argued that truly reusable component libraries require the ability to adapt components by extending them with new functionality, and to compose components via first-class plugs. Object-oriented programming and abstract state machines each provide one of these two keys to successful composition. The view construct aims at providing both of these keys.

Work is currently underway on a complete design and implementation of views in the context of Funnel, an implementation language for functional nets. Only future experience will tell whether the two keys are sufficient for the construction of truly reusable component libraries. That's why the title of this paper still ends in a question mark.

References

1. D.G. Bobrow, K. Kahn, and G. et al. Kiczales. CommonLoops — merging Lisp and object-oriented programming. In *Proc. OOPSLA'86*, pages 17–29, 1986.
2. Don Box. *Essential COM*. Addison Wesley, 1998.
3. Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.
4. Craig Chambers. Object-oriented multi-methods in Cecil. In *Proc. ECOOP*, 1992.
5. Kung Chen, Paul Hudak, and Martin Odersky. Parametric type classes. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 170–181, June 1992.
6. Robert Englander. *Developing Java Beans*. O'Reilly & Associates, 1997.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
8. Cordelia Hall, Kevin Hammond, Simon Peyton Jones, and Philip Wadler. Type classes in Haskell. In *Proc. 5th European Symposium on Programming*, pages 241–256, 1994. Springer LNCS 788.
9. William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proc. OOPSLA*, pages 411–428, September 1993.
10. Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *Proc. ECOOP*, volume 707 of *Springer Verlag, Lecture Notes in Computer Science*, pages 36–56, 1993.
11. Paul Hudak, Simon Peyton Jones, and Philip Wadler. Report on the programming language Haskell: a non-strict, purely functional language, version 1.2. *SIGPLAN Notices*, 27(5), May 1992. more recent versions at `www.haskell.org/definition`.
12. Mark P. Jones. A theory of qualified types. In *Proc. 4th European Symposium on Programming*, pages 287–306, February 1992. Springer LNCS 582.
13. Mark P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, pages 52–61, June 1993.
14. Mark P. Jones. Type classes with functional dependencies. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 230–244. Springer Verlag, March 2000.
15. Stefan Kaes. Type inference in the presence of overloading, subtyping, and recursive types. In *Proc. ACM Conf. on Lisp and Functional Programming*, pages 193–204, June 1992.
16. Ralph Keller and Urs Hölzle. Binary component adaptation. In *Proc. European Conference on Object Oriented Programming*, Lecture Notes in Computer Science. Springer Verlag, 1998.
17. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. of the European Conference on Object-Oriented Programming*, Finland, June 1997.
18. Konstantin Läufer. Type classes with existential types. *Journal of Functional Programming*, May 1996.
19. Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proc. ECOOP*, number 1628 in LNCS, pages 279–303. Springer Verlag, June 1999.
20. Betrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
21. Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In *Software Architectures and Component Technology*. Kluwer, 2000.

22. Tobias Nipkow and Christian Prehofer. Type checking type classes. In *Proc. 20th ACM Symposium on Principles of Programming Languages*, pages 409–418, 1993.
23. Martion Odersky. Programming with variable functions. In *Proc. International Conference on Functional Programming*, September 1998.
24. Martin Odersky. Functional nets. In *Proc. European Symposium on Programming*, number 1782 in LNCS, pages 1–25. Springer Verlag, March 2000.
25. Alan Pope. *The Corba Reference Guide: Understanding the Common Object Request Broker Architecture*. Addison Wesley, 1998.
26. Clemes Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
27. P. Tarr, H. Ossher, W. Harrison, and S. "Sutton Jr.". *N* degrees of separation: Multi-dimensional separation of concerns. In *Proc. ICSE*, pages 107–119, 1999.
28. Philip Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *14'th ACM Symposium on Principles of Programming Languages*, Munich, Germany, January 1987.
29. Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. In *Proc. 16th ACM Symposium on Principles of Programming Languages*, pages 60–76, January 1989.
30. Gio Wiederholt, Peter Wegner, and Stefano Ceri. Towards megaprogramming. *Communications of the ACM*, 1992. November.

XASM– An Extensible, Component-Based Abstract State Machines Language

Matthias Anlauff

GMD FIRST, Kekuléstr. 7
D-12489 Berlin, Germany
ma@first.gmd.de

Abstract. The Abstract State Machine (ASM) [16] approach has already proven to be suitable for large-scale specifications of realistic systems [18,9,22,34]. Due to the fact that the ASM approach defines a notion of *executing* specifications, it provides a perfect basis for a language, which can be used as a specification language as well as a high-level programming language. However, in order to upgrade to a realistic programming language, such a language must – besides other features – add a modularization concept to the core ASM constructs in order to provide the possibility to structure large-scale ASM-formalizations and to flexibly define reusable specification units. In this paper, the language XASM, which stands for *Extensible ASM*, is presented. XASM realizes a component-based modularization concept based on the notion of external functions as defined in ASMs. This paper also briefly describes the support environment of XASM consisting of the XASM-compiler translating XASM programs to C source code, and the graphical debugging and animation tool.

1 Introduction

The Abstract State Machine approach has been and is successfully used to model a large number of case studies including industry-relevant ones. The simplicity of the basic data and execution model of ASMs makes them perfectly suitable as the basis for a language that on the one hand can be used as specification language and on the other hand as a high-level programming language. In this paper, the XASM (*Extensible ASM*)[1] language is presented which aims at providing support for using ASMs as a programming language for producing efficient and reusable programs. There exists a number of other ASM implementations which all implement most of the ASM constructs as defined in the Lipari-Guide [16]. While the realization of the ASM constructs can be seen as the core functionality which must be present in each ASM support system, the difference of an ASM system compared to all others can be characterized by

[1] formerly known as "Aslan"; the name has been changed because of a name conflict with another tool.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 69–90, 2000.

- its efficiency,
- the functionality of its support environment,
- its rule abstraction concept, and
- its interoperability with other languages and systems.

For example, all ASM implementations – including XASM– define some macro structures on top of the core ASM language in order to provide some kind of rule abstraction concept. These additional features are indispensable for managing large formalizations. In the ASM-Workbench [13], for instance, the a special "Rule" construct is introduced being used to assemble ASM specifications from smaller pieces.

Concerning these features, XASM combines the advantages of using a formally defined method with the features of a full-scale, component-based programming language and its support environment.

The paper is organized as follows: In Section 2 an overview of XASM is given. Section 3 introduces the component-based module concept of XASM, in Section 4 the external language interface of XASM is described. In Section 6 the possibility to specify the syntax of input languages using context-free grammar definitions is presented, which is followed by the description of non-standard language constructs defined in XASM in Section 5. Section 7 sketches the support environment of XASM; Section 8 contains concluding remarks and points out future work.

2 Overview of XASM

XASM is an implementation of sequential ASMs focusing on the generation of efficient executable programs simulating the run of the specified ASM. In general, the main design goals of XASM can be given as follows:

- full support of the ASM language as defined in the Lipari-Guide;
- efficient execution of generated executables;
- comfortable animation and debugging of ASM specifications;
- component-based library concept for managing large-scale specifications;
- external language interface for integrating ASM specifications in other systems.

The scenario of building ASM-based programs using XASM is depicted in Figure 1. XASM source files are translated into C source by the XASM-compiler. Additionally, the user can integrate C-sources and -libraries using the external language interface. As described below, XASM introduces a notion of components being stored in a special repository. During the translation process, the XASM-compiler retrieves registry information from the component in order to integrate pre-compiled XASM-components in the current build process. The result of such a build process is a binary being either an executable or a new element of the component library. In either case, the binary contains the ASM algorithms specified in the XASM source files.

Basically, XASM-programs are structured using "**asm** ... **endasm**" constructs each of which containing a list of local function and universe declarations

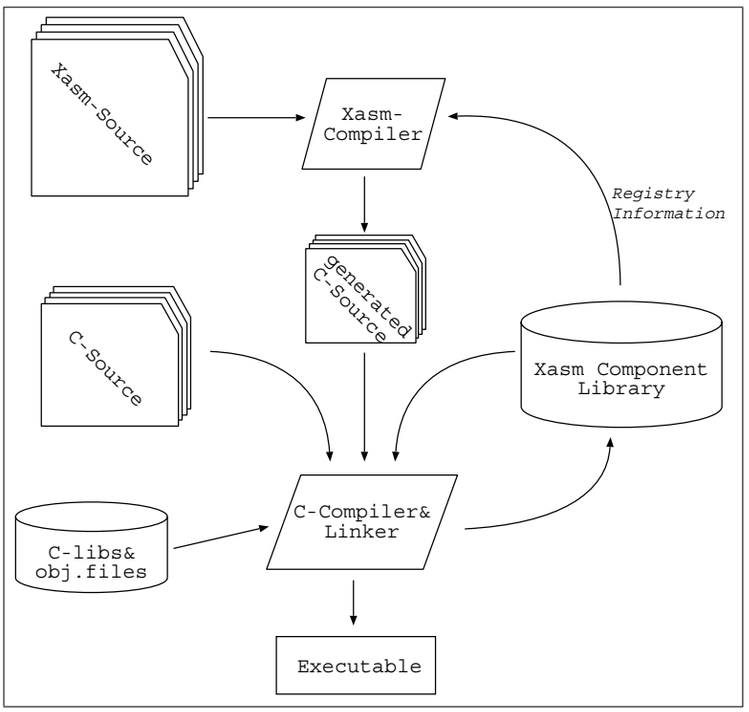


Fig. 1. Building XASM applications

and a list of ASM rules representing a certain part of the overall specification. In general, the structure of an XASM-**asm** is shown in Figure 2. The meta information part contains information concerning the role of the **asm** as a reusable component; this part is described in more detail below.

As defined in the Lipari-Guide, types are not part of the core ASM language. However, because typing has been proven to be very useful to avoid many kinds of errors, in XASM types can be supplied to the declaration of a function and are used to detect static semantics inconsistencies of the formalization.

3 The Basic Structure of XASM Programs: The XASM Component Model

In order to provide the full comfort of a modern programming language, pure ASMs lack a concept of modularization which is indispensable for structuring large-scale formalizations. Macros, which are normally used in the ASM literature to structure large ASM formalizations, only provide limited functionality with respect to the advantages one expects from a module concept. However, macros are a good means for "ASM-programming-in-the-small", but they fail

```
asm A(a_1 : T_1, ..., a_n : T_n) → a_0 : T_0
    ⟨asm meta information⟩
  is
    ⟨universe, function, and subasm declarations⟩
    ⟨initialization rules⟩

    ⟨asm rules⟩
endasm
```

Fig. 2. The general structure of an XASM-**asm**

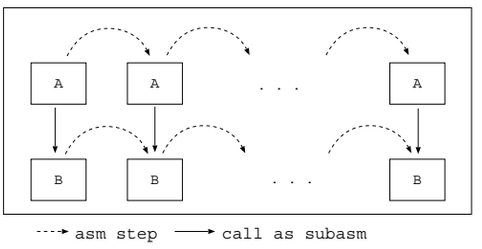
to provide a basis for writing ASM formalizations that can be re-used in other formalizations.

Therefore, XASM uses a more powerful modularization concept which is based on the notion of a *component* as it is used in component-based systems (e.g. [28, 31]).

3.1 The Use Modes of "asm"-Constructs

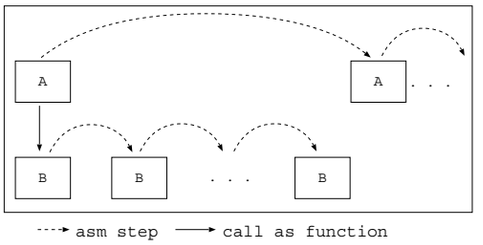
As mentioned above, an XASM formalization is structured using "**asm**...**endasm**" units. In order to explain the relationships that can exist between these units, we will first introduce the possible "use modes": An **asm** can be accessed by other **asms** in either of the following two ways:

If an asm A uses B as **sub-asm**, it means that B – possibly together with arguments, if the arity of $B > 0$ – is used as a *rule* in the body of A. If this rule fires, the rules of **asm** B fire, which may result in updating locations of functions declared in A. The sub-asm-use relation between **asms** may contain cycles; lazy evaluation techniques are used to avoid an infinite number of rules. The call as subasm is illustrated in the figure. The sub-asm B and its parent asm A step simultaneously; formally they can be seen as one single ASM.



Asm A uses B as a **function**, if B is defined as **external function** in A. In this case, B – possibly together with arguments, if the arity of $B > 0$ – is used as a

term in the body of A. Recursion is allowed, so that the function-use relation between **asms** may contain cycles. The call as function is illustrated in the figure. During the run of the function-asm B, its parent A doesn't make any step; from A's point of view B's run happens in zero time. As depicted in the figure, B behaves like a "normal" asm, the iterations shown here are caused by the steps of the B-asm itself.



In each of the above cases, we call A the *parent-asm* of B, if A uses B as sub-asm or as function. In any case, the **asm** must be declared in the parent **asm**. As part of its meta information, an **asm** can be marked as a function or as a sub-asm, so that it can only be used by other **asms** in the specified way. For example, if B and C are **asms** defined as follows

```
asm B(x : Int) → Int
    used as function
  is
    ...
endasm
```

```
asm C(x : Int)
    used as subasm
  is
    ...
endasm
```

then B can only be used as function and C as sub-asm in other **asms**. This is reflected by corresponding declarations of B and C:

```
asm A
  is
    subasm C(x : Int)
    external function B(x : Int) → Int
    ...
endasm
```

Example

A typical situation for using sub-asms is given, when the specification can be split up naturally into several sub-specifications each of which modeling a certain aspect of the overall specification.

```
asm Robot is
    universe ModeValue = {standing, moving}
    subasms Robot_is_standing,
            Robot_is_moving
    function mode → ModeValue
    ...
    if mode = standing then
      Robot_is_standing
    elseif mode = moving then
      Robot_is_moving
    endif
    ...
endasm
```

```
asm Robot_is_standing
    used as subasm
  is
    ...
    mode := moving
    ...
endasm
asm Robot_is_moving
    used as subasm
  is
    ...
    mode := standing
    ...
endasm
```

In this case, the specification introduces the notion of a mode which can be used to structure the formalization. In the example, it is assumed, that the sub-asms update the value of the mode function to some value.

3.2 XASM Components

The declaration of sub-asms and external functions that refer to other **asms** in the specification requires that the existence and functionality of these **asms** is known at specification time. This is comparable to a static module concept and is useful for defining sub-parts of one specific formalization. In order to be "component-based" like announced above, the module concept must be enriched with some other features allowing a more flexible and comfortable definition of reusable units.

For example, consider the following **asm** that may be used in the context of a programming language semantics specification. It checks, whether a given variable is defined in the current block, or in one of the parent blocks. The information whether a variable is defined in a certain block is stored in the ASM function *DeclTable*; the block structure is stored in the function *ParentBlock* mapping blocks to its corresponding parent blocks:[2]

[2] the meaning of the **return** rule is explained later

```
asm check_blockvar(block : Str, var : Str) → Bool
    used as function
    accesses functions DeclTable(Block : Str, var : Str) → _ ,
                       ParentBlock(Block : Str) → Str
  is
    function current_block ← block
    if DeclTable(current_block, var) ≠ undef then
      return true
    else
      current_block := ParentBlock(current_block);
      if current_block = undef then
        return false
      endif
    endif
endasm
```

Note, that these function works correctly without recursively calling itself; it iterates until no update changes the internal state of the asm.

The "**accesses**" construct is used to specify the functions the **asm** expects from its parent **asm**. Now, with this additional meta information, the **asm** can be regarded as a *component*, because its provides information necessary to be processed as stand-alone unit. The **asm** can be separately compiled and put into the XASM component library; other formalization can reuse it provided that they declare the required functions.

Besides the "**accesses**" construct, which allows to *read* the locations of the corresponding functions provided by the parent **asm**, the XASM "**updates**" construct marks the corresponding function as readable *and* writable for the sub-asm or ASM function. In the previous example, the *mode* function must be marked as "updated" the two sub-asms, because it is updated in the body of each of them:

```
asm Robot_is_standing
    used as subasm in Robot
    updates function
      mode → ModeValue
  is
    ...
    mode := moving
    ...
endasm
```

```
asm Robot_is_moving
    used as subasm in Robot
    updates function
      mode → ModeValue
  is
    ...
    mode := standing
    ...
endasm
```

Like the accessed functions, the updated functions must be declared in the parent **asm**. In order to avoid repetitions in the source code, the notation "**used as subasm in** A" can be used as an abbreviation of accessing all functions and universes declared in A except those that are explicitly marked as "updated" by the sub-asm (analogously for asms that are used as functions).

Besides functions, sub-asms can also be contained in the "accesses"-list of an asm-component. The accessed sub-asms are used in the rule section of the asm as if they have been declared locally.

3.3 Gluing of XASM Components

In order to provide a high degree of flexibility in interconnecting XASM-components, it is possible to define local derived functions using so-called "**with**" definitions. For example, a **asm** A wants to use the *check_blockvar* as introduced above, but A doesn't declare a function named "$DeclTable$" as it is required by the *check_blockvar* **asm**. However, the $DeclTable$ must be somehow expressible using existing functions in A. The "with"-statement can be used to provide the called **asm** with the necessary function declaration, as illustrated in the following example:

```
asm A is
    ...
    function currentmodule → Str
    function SymTable(mod : Str, block : Str, v : Str) → Int
    external function check_blockvar(b : String, v : Str) → Bool
            with DeclTable(b : Str, v : Str) ==
                    SymTable(currentmodule, b, v)
    ...
endasm
```

In a similar way, accessed sub-asms can be specified in the context of a "with" statement.

Using XASM components together with this kind of gluing mechanism provides a powerful means to structure large specifications using smaller and reusable units.

3.4 Informal Semantics of "Accesses" and "Updates" Declarations

The semantics of the "accesses" and "updates" declaration in **asms** depends on whether the **asm** is used as a sub-asm or as a function. In the following, the semantics of these constructs in each of these cases is explained briefly.

Accessed and updated functions in Sub-Asms. If an **asm** B is used in A as a sub-asm, the accessed and updated functions in B are directly linked to the corresponding functions in A. That means that if B updates a function declared in A, the update is visible for both A and B in the subsequent step. This can be done in this way, because A and B step simultaneously; the rules of B are regarded as part of the rules of A. Similarly, if B accesses a sub-asm C, then firing C in B has the same effect than firing C in A.

Accessed and updated functions in External Functions. The more complicated case is given when an **asm** B being used as an external function in A accesses and updates functions declared in A. Due to the fact that A doesn't make any step during the run of B, rules in B updating dynamic functions declared in A are actually not performed from B's point of view. Therefore, in XASM the semantics of updated functions in external asm functions is defined in a way, that these kind of unintuitive behavior is avoided:

- For each function f being marked as "updated" a local function with the same name is (internally) declared in B;
- this local function is initialized with the values of the original function in A;
- during the run of B, the functions marked as "updated" can be accessed like any other local function in B;
- on termination of B the updated locations of these function are propagated to the original function declared in the parent **asm** A.

This ensures, that *all updates* of an "updates" function are accessible in B, and that only the *last updates* are forwarded to the parent A. In general, the updates of functions declared in A and updated in B are treated as being part of the update set of A's current step. As a consequence, multiple invocations of B in the same step of A do not influence each other.

Consider the following – somewhat artificial – example:

```
asm A is
    function f(x : Int) → Int
    function v → Int
    external function B → Int
    ...
    v := B
    ...
endasm
```

```
asm B → Int
    used as function
    updates function f(x : Int) → Int
  is
    function i ← 0
    function r
    if i < 3 then
      r := f(0)
      f(0) := i
      i := i + 1
    else
      return r
    endif
endasm
```

In each step of the run of **asm** B the value of the updated function $f(0)$ is updated with a new value. The semantics of the "updates" declaration in XASM ensures that *all* updates of f are accessible in B and that only the *last* update of f in B is propagated to the parent **asm** A. In this example, the update $f(0) := 2$

is propagated to A, while all other updates occurring in the "internal" steps in B[3] have only local effects in B.[4]

3.5 Access Modes of External Functions

In order to allow different kinds of accesses to external entities, external functions can be declared either as "monitored" or as "output" functions. In the first case, the external function can be read, but not written (e.g user input), in the second case, the external function can be written, but not read (e.g. output channels like stdout and stderr).

As a restriction, an external function can be either a monitored *or* an output function, not both. If a function would have both modes, reasoning about the values of that function would require special case distinctions: If a location of such a function is updated in one step of the ASM by means of an update rule, it cannot be guaranteed that the location has the updated value in the next step, because the environment might have changed it in the meantime.

In the following example, the 0-ary function *error* is defined as an external function with "output" access mode; it is used in the parent **asm** for displaying an error message on "stderr" and for setting an ok_flag to false.

```
asm A is
    relation checkok
    external [output] function error → Str
                 with ok_flag == checkok
    ...
    error := "..."
    ...
endasm

asm error → msg : Str
    used as function
    updates relation ok_flag
  is
    use stdio
    stderr := msg
    ok_flag := false
endasm
```

The value that is used for updating the external function can be accessed using the named result parameter *msg*. The **use** construct includes pre-defined header files containing function declaration that are in this case used to declare the external functions *stdout*, *stdin*, and *stderr*.

[3] in step 1: $f(0) := 0$; in step 2: $f(0) := 1$

[4] that means that in the second step the update $r := 0$ is performed in B, in the third step $r := 1$

If no access mode is specified in the declaration of an external function, the mode "monitored" is assumed.

3.6 Scoping

In the context of use-relations between **asms**, XASM distinguishes between the *parent-asm* and the *caller-asm*:

> The **parent-asm** of an **asm** B is the **asm** where B is declared (either as sub-asm or as external function), while the **caller-asm** is the **asm** where the call actually takes place.

In the easiest case, parent and caller are the same, as in the above example: **asm** *error* is declared in and called by **asm** A. In the following example, this is not the case:

```
asm A is
    relation checkok
    external [output] function error → Str
                with ok_flag == checkok
    subasm B
    ...
    B
    ...
endasm

asm B
    updates function error → Str
  is
    error := "..."
endasm
```

```
asm error → msg : Str
    used as function
    updates
      relation ok_flag
  is
    use stdio
    stderr := msg
    ok_flag := false
endasm
```

Here, A is the parent-asm and B the caller-asm of **asm** *error*. As a consequence, the exported relation ok_flag is taken from the parent-asm, rather than from the caller-asm. That means, that the update of *error* in B has the consequence that the *checkok* relation is updated in A. This distinction has been made, in order to completely abstract from the actual realization of exported and accessed functions and sub-asms. In this case, B doesn't need to "know" that *error* is an external function.

The scoping rule for XASM-**asms** is similar to static scoping in programming languages and can be summarized as follows:

> *Exported and accessed functions and sub-asms of an* **asm** B *are always taken from the* **asm** *where* B *has been declared either as external function or as sub-asm.*

3.7 Returning Values From External Functions

As already frequently used in the examples in this document, the **return** construct is used to specify the return value of a monitored external function. In terms of ASMs, "**return** " is realized as follows: In each **asm** B that is used as monitored function, a 0-ary dynamic function B_result is declared and initialized with no_result, a specific element of the superuniverse. Let R be the rule of B as defined by the user, then the internally used rule representing the body of B is given by the following conditional:

if $B\_result = no\_result$ **then**
  R
endif

In other words, updating B_result with value different from the special no_result element directly forces the **asm** to terminate. The notation "**return** t" in an **asm** B is then simply an abbreviation for the update "$B\_result := t$".

4 The XASM External Language Interface

In order to integrate ASM algorithms into other applications, XASM defines an external language interface. In the current version, this interface is implemented for the connection of XASM programs with programs written in the C language. Interfaces to other language, like Java, are in preparation.

In principle, there are two alternatives how the interconnection to the external application can be realized:

- C-functions are used to implement external ASM functions, or
- XASM-**asms** are called from the C main program.

In the first case, the main control of the application is handled by the XASM-part of the system, while in the second case the C-application has the main control. This is also reflected by the definition of the "main" C function: in the first case it is contained in the XASM-part, in the second case, the C-part must provide it. The corresponding interfaces of XASM for these two alternatives are explained in the following.

4.1 External C-Functions

In the previous section we have shown, that external function can be specified in XASM using the **asm** construct. Alternatively, external functions can be implemented in C. The corresponding XASM-declaration is given as follows:

external `"C:`*c_name*`"` [*access_mode*] **function**
  $xasm\_name(a_1 : T_1, \ldots, a_n : T_n) \rightarrow T$

The *c_name* specifies the name of the C function; it can be omitted, if it is equal to the XASM-name of the function. Depending on the access mode of the XASM-function, the corresponding C-function prototypes differ slightly:

– If the access mode is "monitored", the C-functions are defined as follows:
  `ASMOBJ` *c_name*`(ASM a, int argc, ASMOBJ* argv);`
  where "ASMOBJ" is the C-type representing elements of the superuniverse; "ASM" is a C-struct containing information related to parent **asm**. These types are specified in the header file "xasm.h" which must be included in those files containing external function implementations. The arguments of the function call can be accessed via the "argv" field using "argc" as argument count. The first argument, argv[0], always contains the name of the corresponding XASM-function as string element. The result of the external function that is accessible in the calling **asm** is returned by the C-function as "ASMOBJ".
– If the access mode is "output", the C-functions are defined as follows:
  `void` *cname*`(ASM a, int argc, ASMOBJ* argv, ASMOBJ val);`
  In this case, additionally the value that is used in the update representing the call of the external function can be accessed using the "val" parameter.

As an example, the following C-code contains the implementation of the "stderr" function previously used in one of the examples:

```
void xasm_stderr (ASM a, int argc, ASMOBJ* argv, ASMOBJ val) {
  if (argc != 1) {
    error("wrong # args for external function '%s'.\n",
          c_stringvalue(argv[0]));
    return;
  }
  fprintf(stderr,"%s",str_obj(val));
}
```

The XASM-library function "str_obj" returns the string representation of an ASMOBJ. The corresponding declaration of the external C-function in an asm has the following format:

external `"C:xasm_stderr" [output]` **function** *stderr* → *String*

4.2 Embedding XASM-Programs in C-Applications

If the XASM-part of a system should provide services for a C-based application, the main **asm** and all sub-asms of it can be called from the C-code. In this case, the XASM-compiler must be invoked with a special option that prevents the generation of the "main"-function.

Before any of the **asms** can be invoked, the XASM-part must be initialized. For this purpose, the generated C-code defines the function "asm_main" as follows:

```
int asm_main(int argc, char **argv);
```

The arguments to this function are given as strings which are parsed and transformed to corresponding "ASMOBJs". The actual invocation of the main **asm** can be made using the C-function "run_mainasm":

```
ASMOBJ run_mainasm();
```

This kind of embedding is actually used in the implementation of the XASM-compiler itself: the static semantics check is carried out by an algorithm specified in XASM.

A number of external C-functions are already integrated into the runtime system. For example, external functions to communicate using UNIX-Sockets, string manipulation functions, file access functions etc. .

5 Non-standard Language Constructs of XASM

Besides the implementation of the ASM core constructs, XASM provides a number of useful extensions that can all be directly mapped to the original ASM constructs. In the following, some of these extensions are described briefly; a full version of the language specification is in preparation and will be available shortly.

5.1 Constructor Terms

XASM provides the possibility to define and use *constructor terms*. The concept of constructor terms can be mapped to the ASM core language as follows: According to [17] each of the function names contained in the vocabulary V of an ASM may be marked as *relational* or *static*, or both. In addition, we allow static functions to be marked as *constructive*. Let F_c be the set containing all functions in V marked as constructive, $F_c \subseteq V$. Let $f_c \in F_c$, arity of $f_c = n$, then the following conditions hold for all states A of the ASM:

(i) $\forall t_1, \ldots, t_n, \mathrm{V}al_A(t_i) \neq$ **undef** $, 1 \leq i \leq n \bullet$
$f_c(t_1, \ldots, t_n) \neq$ **undef**

(ii) $\forall g_c \in F_c,$ arity of g_c is m; $\quad t_1, \ldots, t_n, \mathrm{V}al_A(t_i) \neq$ **undef** $\bullet$
$f_c(t_1, \ldots, t_n) = g_c(s_1, \ldots, s_m) \Leftrightarrow$
$f_c = g_c \wedge n = m$
$\wedge \mathrm{V}al_A(t_i) = \mathrm{V}al_A(s_i), 1 \leq i \leq n$

where $\mathrm{Val}_A(t)$ stands for the evaluation of term t in state A of the ASM. Informally speaking that means that each constructive function is (i) defined at all locations and that (ii) the content of each location is a unique element of the superuniverse w.r.t. the set of locations of all constructive functions. If $f_c \in F_c$, then f_c is called a *constructor*, and the terms $f_c(t_1, \ldots, t_n)$ are called *constructor terms*. In XASM, the declaration of a constructor is part of the function declarations, for example

constructor $nil, cons(\_, \_)$
universe $BinTree = \{empty, children(l : BinTree, r : BinTree)\}$

introduces the constructors *nil*, *cons*, *empty*, and *children*, where terms constructed using the latter two constructors are elements of the universe $BinTree$.

XASM also provides pattern matching functionality like it is used in many other languages. Syntactically, pattern matching terms are used as condition terms in conditionals, e. g.: [5]

```
if b =~ children(&l, &r) then
   R(&l, &r)
else
   ...
endif
```

There are three kinds of pre-defined, commonly-used constructors in XASM: sets, sequences, and tuples. These constructors are specified using their usual representation: $\{x_1, \ldots, x_n\}$ for sets, $[x_1, \ldots, x_n]$ for sequence, and $(x_1, \ldots, x_n)$ for n-tuples. For sequences, the notation $[H|T]$ can be used in pattern matching terms for accessing head and tail of a sequence.

5.2 Regular Expressions

In practice, strings are widely used as a common data format for exchanging information between different systems. XASM therefore provides a special kind of pattern matching based on *regular expressions* as they are used in UNIX (e.g. in the "sed" program) as well as in many scripting languages like Perl [33] and Tcl [26]. The regular expression pattern matching is invoked using the =~ operator like for pattern matching with constructor terms. If both operands of the =~ operator are strings, then the right operand is interpreted as regular expression and the left operand as string being match against the regular expression. For example, the regular expression pattern matching expression

```
s =~ '^[A-Z]'
```

evaluates to true, if s is a string starting with a capital letter.

In regular expressions, parenthesis "\(..\)" can be used to mark certain parts of the expression that correspond to sub-strings of the left-hand-side string, if the pattern matching has been successful. For that, Xasm provide a special form of regular expression pattern matching: If the left-hand-term of a pattern matching expression evaluates to a string object s and the right-hand-term evaluates to a tuple the first argument of which represents a string object r and the remaining arguments are pattern matching variables $\&v_1, \ldots, \&v_n$, the string s is matched against the regular expression r and the sub-matches are put into the pattern matching variables $\&v_1, \ldots, \&v_n$, if the match has been successful.

[5] XASM uses a special syntax for pattern matching variables and equality symbol

if `"AnyString"` $=\sim$ (`'^\(.\)\(.*\)$'`, $\&hd, \&tl$) **then**
  ...
endif

In this example, the regular expression contains two sub-matches, the first one matches the string `"A"`, the second one the string `"nyString"`.[6] The submatches can be accessed in the then-part of the conditional rules as values of the pattern matching variables $\&hd$ and $\&tl$.

5.3 The "Once"-Rule

A common situation occurring in ASM formalizations is that certain rules should fire only once during the run of an ASM. Normally, one has to introduce extra functions in order to ensure this behavior. XASM introduces a "once" rule for these situations:

if once R_1
else R_2
endif

which is – in terms of ASMs – equivalent to the conditional rule

if $once(n)$ **then**
  R_1
  $once(n) := false$
else
  R_2
endif

where n is a unique number representing the n'th occurrence of a "once"-rule in the ASM, and *once* is a one-ary dynamic relation initialized with **true** for all these numbers n. The notation "**once** R" is an abbreviation for

if once R **else skip endif**

6 Grammar Definitions in XASM

Historically, XASM has been developed as underlying ASM implementation for *Montages*, a semi-visual method for specifying the syntax and semantics of programming languages, see [21,2,3]. As a consequence, the support for programming language related features has been integrated into the XASM language as a means to extend the original syntax with domain-specific constructs. The syntax and semantics of these extensions can be specified using the Montages method together with tool support environment Gem-Mex which translates user-defined language definitions into XASM-code.

[6] Single quotes are used for the regular expressions in order to prevent interpretation of special symbols (like "\") in the string. To understand the example: `^` stands for the beginning, `$` for the end of a string, the dot represents any character, and the asterisk stand for zero or more occurrences of the preceeding expression.

The grammar definitions used for the translation of Montages-specification can also be used directly in XASM. For that purpose, **nonterm** and **token**-declarations can be given in XASM, resulting in the generation of a parser for the specified language. As an example for using grammar definitions in XASM, Figure 3 contains the specification of a parser that accepts empty XML tags. The generated C-function can be accessed using an external function returning the root node of the parse tree being constructed during parsing.

```
asm xmlparser(inpfile) is
    use syntax
    token Ident = "[A-Za-z][A-Za-z0-9_]*"
    startnonterm Elements
    nonterm Elements[Element]
      base /* empty */
      cont Elements Element
    endnonterm
    nonterm Element = ElementEmpty endnonterm
    nonterm ElementEmpty ::= "<"Ident Attributes"/ >";
    endnonterm
    nonterm Attributes[Attribute];
      base /* empty */
      cont Attributes Attribute
    endnonterm
    nonterm Attribute ::= Ident"="String_token;
        lhs ← (Ident.Name, String_token.Name)
    endnonterm
    external "C" function parse_Elements(filename : String) → Elements
    function RootNode → Elements
    RootNode := parse_Elements(inpfile);
endasm
```

Fig. 3. A grammar specification in XASM for parsing empty XML Elements

7 The XASM Support Environment

The support environment of XASM consists of the XASM-compiler, the runtime system and the graphical debugging and animation interface. In this paper, these tools are only sketched briefly, a more detailed description will be contained in the XASM user manual which is currently under development. A description of the graphical animation tool is also contained in [2,3].

7.1 The XASM-Compiler and Runtime System

As already mentioned in Section 2, the XASM-compiler `xasmc` translates XASM source code into C-code implementing the executable version of the ASMs specified in the XASM source files. The syntax analysis part is implemented in C using "lex" and "yacc" for generating scanner and parser code. The type checking part is implemented in XASM itself using the C-interface as introduced above.

The XASM runtime system implements the core functionality of the XASM language. In here, all algorithms and data structures are realized being used to transform an ASM into an executable program. At the heart of the runtime system is the implementation of update and access functionality for ASM functions. For that, a hashing mechanism is used to provide optimized access to values of ASM functions. The runtime system also contains garbage collection facilities, which is indispensable, if ASM algorithms are used for continuous control systems, as described in [5].

7.2 The XASM Graphical Animation and Debugging Interface

In order to be able to animate and/or debug the XASM program, a graphical animation and debugging tool has been realized that enables to stepwisely execute the ASM, to trace updates that has been performed in each step, and to view function values in each step. In case a grammar has been specified as input format for the XASM program, a special kind of graphical animation window can be used to display function values that refer to node in the parse tree. Figure 4 shows a screen-dump of a debugging session. Additionally, an integrated design environment, incorporating the graphical user interface is currently under development.

7.3 The XASM-LaTeX Package

As an additional support feature, an LaTeX-package "asm.sty" is defined for typesetting XASM specifications. The LaTeX-files can directly be used as input file to the XASM-compiler, so that no additional work is necessary to produce a high-quality documentation from a running XASM specification. The XASM code parts in this document are produced using the asm style being realized based on the LaTeX "program" style as defined in [14].

8 Conclusion

In this paper, the ASM based language XASM has been presented focusing on the additional features provided by the language with respect to the ASM core concepts as defined in the Lipari Guide. A novel concept for structuring ASM specifications based on the notion of components has been presented. This concept perfectly fits into the basic model of the ASM approach, because it allows to choose the level of abstraction for describing that fits best to a given problem without regarding technical constraints. Furthermore, this concept allows

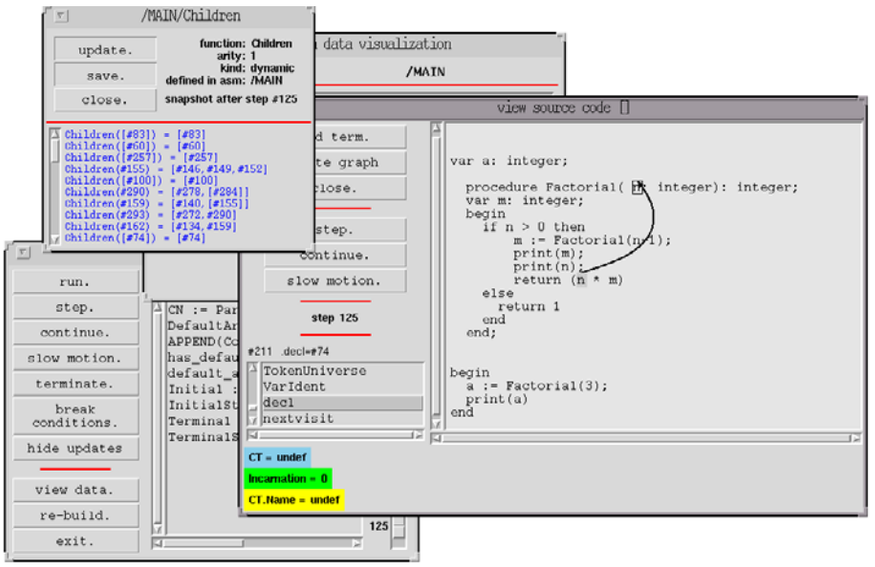


Fig. 4. Snapshot of a XASM-debugger session

efficient development cycles, because asm's can be designed as reusable components by exactly specifying what is expected from the environment. An algebraic view of a similar structuring concept has been given by Wolfgang May in [25]. He tries to formalize the notion of "hierarchical ASMs" which is used in early papers of Gurevich and Börger. Later in [10] special cases of the May's concepts are proposed for ASM composition. These cases as well as May's general notion of "hierarchical ASMs" together with means for data encapsulation are directly supported by XASM.

The language presented in this paper is fully implemented. The system is used as the basis for the Montages/Gem-Mex, where generated XASM code is translated into an interpreter for the language specified using Montages. As a case study the full specification of ANSI C has been entered by Wuwei Shen [19]. Other case studies are currently under development, see for example an application to microprocessor simulation in [32] and the application of XASM as gluing code in legacy systems [4].

As next steps, the XASM compiler, runtime system and graphical support environment will be further optimized. Also, a concept how tools for the (automatic) verification, model checking and analysis of the ASM formalizations can be integrated into the system is currently under development and will be part of the support system in future versions of the tool. Furthermore, the connection to repository systems will be subject to future considerations concerning the XASM-language. For example, certain ASM functions may be marked as "persistent", meaning that the values of the locations of these functions are stored

in the repository system, so that they can be access the next time the XASM is executed. This kind of extension is currently part of work carried out in an industry-based research project running at GMD In this project, it is currently considered to use XASM for formulating certain consistency check algorithms occurring in the context of this project.

Additional applications outside the ASM area are possible, since ASMs can be considered as an instance of so called *transitions system models*, which form as well the basis for other popular formalisms such as UNITY [11], TLA [23], SPL [24] and the SAL intermediate language [29]. As mentioned in Section 6 the XASM system is integrated with the Montages method so that new or alternative constructs can be easily supported by XASM. Using Montages, both syntax and semantics of new or alternative XASM constructs can be developed in an integrated development environment called Gem-Mex. Such an extensible system architecture allows the XASM tool to be tailored to one of the above-mentioned formalisms based on transition systems.

An adapted version of XASM would be especially useful in a context like the SAL architecture [7], where various other tools are integrated for tasks like theorem proving and model checking. The development cycle supported by the tools integrated with SAL could be extended with a tool like XASM for debugging and animation of the transition system under consideration. After the developer gains confidence in his specification by debugging and animating them with XASM, other tools can be used to further analyze the specification. As a result of the further analysis, the original specification typically undergoes major changes. To avoid simple errors introduced by these changes, the specification can again be debugged and animated using XASM before it is passed to the other tools.

We plan to adapt XASM to architectures like SAL in order to make it accessible to a large base of users. We can thus adapt the Montages method to formalisms like SAL by providing a language semantics with SAL. A specific language may then be developed so that the SAL semantics of programs written in that language can be easily analyzed using the available tools. The concrete syntax of such a language can be adapted to the terminology of domain experts using the language to describe their system. Examples of using Montages to develop languages for special domains are given in [20,1,4].

Acknowledgments. I very much thank Philipp W. Kutter and Alfonso Pierantonio for their collaboration and fruitful discussions which had a great influence in the design and implementation of XASM. I also like to thank Yuri Gurevich and Egon Börger for their interest in my work presented in this paper and for always helping me solving the right problems. My dearest special thanks go to Asuman Sünbül. Her work in the field of component-based software engineering has very much influenced the component model presented in this paper.

References

1. M. Anlauff, A. Bemporad, S. Chakraborty, P.W. Kutter, D. Mignone, M. Morari, A. Pierantonio, and L. Thiele. From ease in programming to easy maintenance: Extending DSL usability with Montages. Technical Report 84, Institute TIK, ETH Zürich, December 19999.
2. M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.
3. M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In A.Zamulin D.Bjoerner, M.Broy, editor, *Perspective of System Informatics*, volume 1755 of *LNCS*, pages 40 – 53, 1999.
4. M. Anlauff, P.W. Kutter, A. Pierantonio, and Asuman Sünbül. Using domain-specific languages for the realization of component composition. In *Proceedings Formal Approaches in Software Engineering FASE00*, LNCS, 2000.
5. M. Anlauff and A. Sünbül. An ASM specification of an elevator control system. 1999.
6. M. Anlauff and A. Sünbül. Software architecture based composition of components. In *GI-Workshop Sicherheit und Zuverlässigkeit software-basierter Systeme*, 1999.
7. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Muñoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, June 2000. to appear.
8. J. A. Bergstra and P. Klint. The ToolBus coordination architecture. In Ciancarini and Hankin [12], pages 75–88.
9. E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.
10. E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editor, *Gurevich Festschrift CSL 2000*, LNCS. Springer-Verlag, 2000. to Appear.
11. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
12. Paolo Ciancarini and Chris Hankin, editors. *Coordination and models, Proceedings of the first international conference, Cesena, Italy*, number 1061 in LNCS. Springer Verlag, 1996.
13. G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
14. Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Tools and Techniques for Computer Typesetting. Addison-Wesley, Reading, MA, USA, second edition, 1994.
15. F. Griffel. *Componentware*. dpunkt.verlag, 1998.
16. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
17. Y. Gurevich. May 1997 Draft of the ASM Guide. Department Technical Report CSE-TR-336-97, University of Michigan, 1997.
18. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.

19. J.K. Huggins and W. Shen. The static and dynamic semantics of C. In *Local Proceedings of ASM2000*, TIK Report Nr. 87, 2000.
20. P. W. Kutter, D. Schweizer, and L. Thiele. Integrating formal domain-specific language design in the software life cycle. In *Current Trends in Applied Formal Methods*, LNCS. Springer, October 1998.
21. P.W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
22. P.W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
23. L. Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, 1994.
24. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification*. Springer-Verlag, New York, NY, 1992.
25. W. May. Specifying complex and structured systems with evolving algebras. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97: Theory and Practice of Software Development*, number 1214 in LNCS, pages 535 –549, 1997.
26. John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.
27. J.-G. Schneider and O. Nierstrasz. Scripting: Higher-level programming for component-based systems. In *OOPSLA 1998*, 1998. Tutorial.
28. Jean-Guy Schneider and Oscar Nierstrasz. Components, scripts and glue. In Leonor Barroca, Jon Hall, and Patrick Hall, editors, *Software Architectures – Advances and Applications*, pages 13–25. Springer, 1999.
29. N. Shankar. Symbolic Analysis of Transition Systems. In *This volume*.
30. Asuman Sünbül. *Architectural Design of Evolutionary Software Systems*. PhD thesis, Technical University Berlin, 1999. in preparation.
31. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, N.Y., 1998.
32. J. Teich, P.W. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using asms. In *This volume*.
33. Larry Wall and Randal L. Schwartz. *Programming Perl*. O'Reilly Associates, Inc., Sebastopol, CA, 1990.
34. C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

Generic Facilities in Object-Oriented ASMs *

A. V. Zamulin

Institute of Informatics Systems
Siberian Branch of the Russian Academy of Sciences
630090, Novosibirsk, Russia
e-mail: zam@iis.nsk.su
fax: +7 3832 323494
phone: +7 3832 396258

Abstract. Facilities for defining generic object types, generic type categories, generic functions and generic procedures in an object-oriented ASM are described in the paper. These facilities permit one to specify algorithms over complex data structures abstracting both from the type of the structure components and the structure itself. The use of the facilities is demonstrated by the specifications of some important parts of Standard Template Library for C++.

Keywords: ASM, object types, object categories, generic specifications.

1 Introduction

Object-oriented ASMs as a variant of traditional ASMs [1,2] are formally introduced in [3]. They permit the specification of a dynamic system in terms of mutable and constant objects belonging to different object types. Object types are partially ordered according to a type–subtype relationship which serves for modeling inheritance. Unfortunately, the technique described in the paper does not provide facilities for the specification of generic object types and generic algorithms. At the same time, such facilities are highly needed if one wishes to write reusable specifications.

As a typical example, let us consider the problem of the specification of Standard Template Library (STL) for C++ [4] which will be used in the paper as a running example. Currently the semantics of the library components is given informally as a set of requirements presented partly in English and partly as C++ program fragments. As a result, the semantics remains incomplete and imprecise, depending heavily on reader's (and library implementor's) intuition and knowledge of C++. Therefore, a formal description of STL independent of a particular programming language is highly needed.

STL is based on the notion of *container* which is a data structure consisting of a number of elements of the same type. Several container classes are defined

* This research is supported in part by Russian Foundation for Basic Research under Grant 98-01-00682.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 91–111, 2000.

in STL: vectors, lists, deques, sets, multisets, maps, and multimaps. Other container classes can be defined if needed. Each container class is parameterized by the component type. Thus, for each data structure one can write an algorithm abstracting from the component type. This provides the first level of genericity typical of C++.

To abstract from the container's structure, STL introduces a notion of *iterator* which is a generalization of the pointer notion. Iterators are grouped into different iterator categories providing abstract data-access methods. There are categories of input iterators, output iterators, forward iterators, bidirectional iterators, and random-access iterators. Iterator categories build a hierarchy. This means that a forward iterator is also an input iterator and an output iterator, a bidirectional iterator is also a forward iterator, and a random-access iterator is also a bidirectional iterator. Algorithms can now be written in terms of iterators abstracting from a concrete data structure. Most important here is the fact that an algorithm requiring, say, an input iterator can also use a forward or bidirectional or random-access iterator. This provides the second level of genericity.

One of the ways of the formal STL definition is the use of classical algebraic specifications whose advantages are sound mathematical foundation and existence of specification languages and tools. Taking into account the generic nature of the data structures and iterator categories, a specification language like Spectrum [5] providing parameterized sorts and sort classes can be used for this purpose. However, the notions of iterator and container subsume a notion of *state* which can change when containers are searched or updated. The modeling of the state by classical algebraic specifications involves extra data structures representing the state which must be explicitly passed to a function as argument and yielded as result. Algebraic specifications are also a poor tool for describing a data structure with an arbitrary order of component insertion and deletion. This leads to very complex specifications with problems of describing the differences between different types of containers.

At the same time it seems very natural to consider containers and iterators as objects possessing state and to define container classes and iterator classes as generic object types parameterized by the component type. To define formally iterator categories and thus represent the hierarchy of iterator classes, we need a notion of *type category* similar to that of sort classes of Spectrum and a notion of *generic type category* absent in conventional algebraic specifications. Thus, the adaptation and extension of generic facilities of algebraic specifications for enhancing object-oriented ASMs are the main tasks of this paper.

The paper is organized in the following way. Concrete object types defining sets of states of potentially mutable objects and object-oriented dynamic systems generalizing the communities of object states and their transitions are introduced in Section 2. Unconstrained generic object types are defined in Section 3. Generic vector type and generic list type as typical representatives of Standard Template Library are specified in Section 4 and Section 5, respectively. Object type categories permitting the classification of object types on the base of their operations are introduced in Section 6. Generic object types, generic type cate-

gories and generic functions constrained by object type categories are defined in Section 7. Some related work is discussed in Section 8, and some conclusions are given in Section 9.

It is assumed that the reader is familiar with the basic ASMs notions which can be found in [1,2]. The familiarity with the formal aspects of object-oriented ASMs [3] is desirable, but not obligatory.

2 Basic Notions

2.1 Object Types

We distinguish between *data* and *objects* and, respectively, between *data types* and *object types*. A data identifies itself and is immutable. An object possesses a unique identifier and state which can be mutable. A data type defines a set of immutable values and operations over them. An object type defines a set of states of a potentially mutable object and a number of methods capable to observe or update the object's state. This distinction between data types and object types requires different methods of their specification.

We consider that data types are specified by means of one of the algebraic specification languages like Spectrum [5], Ruslan [6] or CASL [7]. Let (Σ_0, Ax), where Σ_0 is a signature and Ax is a set of axioms, be the specification of a number of data types. Σ_0 is called the *data signature* in the sequel. An algebra $\mathsf{A_0}$ of this signature is called a *data algebra*. An *object-structured signature*, Σ_{obj}, extends Σ_0 with a set of *object type signatures* [3]. Respectively, an *object-structured specification*, (Σ_{obj}, Ax_{obj}), extends (Σ_0, Ax) with a set of *object type specifications*. Constructing the object-structured specification, we accompany each object type signature with the corresponding axioms. Therefore, the notions of object type signature and object type specification correspond, respectively, to the notions of class declaration and class definition of C++.

The object-structured specification is defined in the following way. Let $OTYPE$ be a set of (object type) names and $O\Phi$ a set of object type specifications. The object-structured specification is then a triple $\langle OTYPE, O\Phi, int^o \rangle$, where int^o is a function mapping $OTYPE$ into $O\Phi$. If $T \in OTYPE$, $ots \in O\Phi$, and $int^o(T) = ots$, then the maplet $\langle T \mapsto ots \rangle$ is the specification of the object type T (in this case we sometimes write that ots is marked with T).

The specification of an object type T in this paper has the following form:

class T = **spec**
[object-type-signature]
{axioms},

where *object-type-signature* is a triple *set-of-attribute-declarations; set-of-mutator-declarations; set-of-observer-declarations*.

An attribute or observer declaration has the following form: *operation-name: operation-profile*. The *operation-profile* is either T or $T_1, \ldots, T_n \longrightarrow T$ where T, T_i are data/object type names indicating the types of attribute/observer parameters (if any) and the result type. A mutator declaration is either just a

mutator name or *mutator-name: mutator-profile*, where *mutator-profile* is a sequence of data/object type names indicating the types of mutator parameters.

Intuitively, a tuple of attributes defines an object's state, an observer is a function computing something at a given object's state, and a mutator is a procedure changing an object's state. Attributes are often called *instance variables*, and observers and mutators are often called *methods* in object-oriented programming languages. All of them are called methods in the sequel.

Notation: in the following examples sets of attribute, observer, and mutator declarations in the object type signature are preceded by keywords **attribute**, **observer**, and **mutator**, respectively.

As an example, let us consider a simple model of memory consisting of locations storing integer numbers. A consecutive number of locations forms a vector which is a structure allowing both sequential and random access to its elements. A location representing a vector element is called a *vector iterator* in the sequel.

Such an iterator can be understood as an object possessing a unique identifier (address) and the attribute *value_stored* updated by the mutator *put_value* and observed by the observer *get_value*. Since we wish to provide both sequential and random access to the vector elements, we define several extra observers as well. As a result, we create the following object type signature:

class VecIt = **spec**
[**attribute** value_stored: Integer;
mutator put_value: Integer;
observer get_value: Integer;
advance, retreat: VecIt;
plus, minus: Nat $\longrightarrow$ VecIt;
difference: VecIt $\longrightarrow$ Nat;
eq, neq, less, greater, leq, geq: VecIt $\longrightarrow$ Boolean]

In the above example, the community of vector elements is represented by the class VecIt (vector iterator) with several methods giving possibilities to move either to the next (*advance*) or previous (*retreat*) element, to jump over several elements forward (*plus*) or backward (*minus*), to calculate the distance between two elements (*difference*) and to compare the element (location) identifiers (*eq, neq, less, greater, leq, geq*). Note that the methods *advance*, *retreat*, *plus* and *minus* are observers since they produce objects and do not update the states of the corresponding objects.

Let $\Sigma' = \langle \Sigma_0, \Sigma_{obj} \rangle$. A Σ'-algebra, A, is constructed by extending a Σ_0-algebra, $\mathsf{A_0}$, with a set of elements, called *object identifiers*, for each object type name T. For example, a set of location addresses can be associated with the object type $VecIt$. We denote by $\mathsf{A_T}$ the set of elements associated with a data/object type T in the algebra A. In addition to the set of object identifiers, for each object type T, a (partial) function $\mathsf{at_T^A} : \mathsf{A_T} \longrightarrow (\mathsf{A_{T'_1}}, \ldots, \mathsf{A_{T'_n}} \longrightarrow \mathsf{A_{T'}})$ is associated in the algebra A with the attribute name $at : T'_1, \ldots, T'_n \longrightarrow T'$ (a function $\mathsf{at_T^A} : \mathsf{A_T} \longrightarrow \mathsf{A_{T'}}$ is associated with the attribute name $at : T'$) declared in the signature of T; such a function is called an *attribute function*. If $\mathsf{id} \in \mathsf{A_T}$,

then $\mathsf{at}^{\mathsf{A}}_{\mathsf{T}}(\mathsf{id})$ is an *attribute* of id. Note that, like in object-oriented languages, the object type is always the first parameter type of its attribute function although it is never indicated in the attribute declaration. An object identifier is used as argument when the corresponding object type is a function parameter type, and an object identifier is produced when the corresponding object type is the result type of a function.

An *object* is a pair $(\mathsf{id}, \mathsf{obs})$ where id is an object identifier and obs is a tuple of its attributes called *object's state*. For example, if id is an object identifier of type VecIt in algebra A, then the pair $\langle \mathsf{id}, \mathsf{value\_stored}^{\mathsf{A}}(\mathsf{id}) \rangle$ is an object with the state $\mathsf{value\_stored}^{\mathsf{A}}(\mathsf{id})$.

To define the interpretation of observer and mutator names, a notion of *dynamic system* is introduced.

2.2 Dynamic System

We discussed above only the functions defined inside the frames of object types. In a more general case, an algebra can possess a number of "stand-alone" functions and constants defined outside the frames of object types. In addition to this, higher-order functions for observing and transforming algebras can be defined. Therefore, we introduce a notion of *dynamic system* possessing a number of states and a number of operations for observing and updating the states.

Generally, the specification of an object-oriented dynamic system is represented by the following tuple:

$$\langle (\Sigma_0, Ax), (\Sigma_{obj}, Ax_{obj}), \Sigma_{din}, (\Sigma_{dep}, Ax_{dep}), (\Sigma_{proc}, Ax_{proc}) \rangle.$$

Its first component is the specification (the signature and axioms) of a number of data types and related functions, and the second component is the specification of a number of object types discussed above. The third component is the signature of a set of *dynamic functions/constants* which can be different in different states of a dynamic system.

A dynamic function is declared as follows:

dynamic function *function-name*: $T_1, ..., T_n \longrightarrow T$;

where $T_1, ..., T_n$ are parameter types and T is a result type. A function without parameters is called a *constant* and declared as follows:

dynamic const *constant-name*: T;

where T is the type of the constant.

Examples.

dynamic const an_iterator: VecIt;

dynamic function matrix: Nat, Nat $\longrightarrow$ VecIt;

Let $\Sigma' = \langle \Sigma_0, \Sigma_{obj} \rangle$, $\Sigma = \langle \Sigma', \Sigma_{din} \rangle$, and A' be a Σ'-algebra. A Σ-algebra A is constructed by extending A' with an element $\mathsf{c}^{\mathsf{A}} \in \mathsf{A}'_{\mathsf{T}}$ for a constant declaration $c : T$ from Σ_{din} and a (partial) function $\mathsf{f}^{\mathsf{A}} : \mathsf{A}'_{\mathsf{T}_1} \times \ldots \times \mathsf{A}'_{\mathsf{T}_n} \longrightarrow \mathsf{A}'_{\mathsf{T}}$ for a function declaration $f : T_1, \ldots, T_n \longrightarrow T$ from Σ_{din}. Such an algebra is called an *instance algebra* or *state*. It represents a number of objects and dynamic functions. An update of an object's state or dynamic function as well as creation

or deletion of an object leads to the transformation of one instance algebra into another. Note that the carrier of A is the same as that of A'.

The forth component of the specification of a dynamic system is the signature and axioms of *dependent functions*. They are declared in the same way as the functions in Σ_{din} with the use of the keyword **depend**. The values produced by these functions depend on the state.

The fifth component of the specification of a dynamic system is the signature and axioms of *procedures* serving for state transformations. A procedure is declared as follows:

proc *procedure-name*: $T_1, ..., T_n$;

where $T_1, ..., T_n$ are parameter types (n can be zero, i.e., a procedure can have no parameter). **Example**:

proc allocate: Nat – *allocation of a number of vector iterators*;

Let now OID be the type of object identifiers isomorphic to natural numbers. This means that there is the sort name OID and two function names, say *nat_to_id* and *id_to_nat* in Σ_0 and the axiom $id\_to\_nat(nat\_to\_id(x)) = x$, where x is a universally quantified variable of type Nat, in Ax. Given a Σ_0-algebra $\mathsf{A_0}$, we assume that all object identifiers are selected from the set of values of this type sequentially so that no object identifier is selected twice. We denote by $\mathsf{DS(A_0)}$ a dynamic system based on a data algebra $\mathsf{A_0}$ and by $|\mathsf{DS(A_0)}|$ the set of instance algebras satisfying the following condition: the restriction of each algebra of $|\mathsf{DS(A_0)}|$ to the data signature equals $\mathsf{A_0}$, i.e., each algebra in $|\mathsf{DS(A_0)}|$ is based on the same data algebra.

Given a dynamic system signature, a data algebra $\mathsf{A_0}$ with a set of object identifiers, and a set of instance algebras $|\mathsf{DS(A_0)}|$, we construct a dynamic system $\mathsf{DS(A_0)}$ by associating:

- with each observer name $b : T_1, \ldots, T_n \longrightarrow T'$ in an object type signature marked with T, a partial function, $\mathsf{b_T^{DS(A_0)}}$, mapping a pair $\langle \mathsf{A}, \langle \mathsf{id}, \mathsf{a_1}, \ldots, \mathsf{a_n} \rangle \rangle$ (a pair $\langle \mathsf{A}, \mathsf{id} \rangle$ for an observer name $b : T'$) to an element $\mathsf{a'} \in \mathsf{A_{T'}}$, where $\mathsf{A} \in |\mathsf{DS(A_0)}|$, $\mathsf{id} \in \mathsf{A_T}$, and $\mathsf{a_i} \in \mathsf{A_{T_i}}$, $i = 1, \ldots, n$;
- with each mutator name $m : T_1, \ldots, T_n$ in an object type signature marked with T, a partial function, $\mathsf{m_T^{DS(A_0)}}$, mapping a pair $\langle \mathsf{A}, \langle \mathsf{id}, \mathsf{a_1}, \ldots, \mathsf{a_n} \rangle \rangle$ (a pair $\langle \mathsf{A}, \mathsf{id} \rangle$ for a mutator name m without a profile) to an update set[1], where $\mathsf{A} \in |\mathsf{DS(A_0)}|$, $\mathsf{id} \in \mathsf{A_T}$, and $\mathsf{a_i} \in \mathsf{A_{T_i}}$, $i = 1, \ldots, n$.
- with each dependent function name $f : T_1, \ldots, T_n \longrightarrow T$, a partial function, $\mathsf{f^{DS(A_0)}}$, mapping a pair $\langle \mathsf{A}, \langle \mathsf{a_1}, \ldots, \mathsf{a_n} \rangle \rangle$ (an algebra A for a function name $f : T$) to an element $\mathsf{a} \in \mathsf{A_T}$, where $\mathsf{A} \in |\mathsf{DS(A_0)}|$ and $\mathsf{a_i} \in \mathsf{A_{T_i}}$, $i = 1, \ldots, n$;
- with each procedure name $p : T_1, \ldots, T_n$, a partial function, $\mathsf{p^{DS(A_0)}}$, mapping a pair $\langle \mathsf{A}, \langle \mathsf{a_1}, \ldots, \mathsf{a_n} \rangle \rangle$ (an algebra A for a procedure name p without a profile) to an update set, where $\mathsf{A} \in |\mathsf{DS(A_0)}|$ and $\mathsf{a_i} \in \mathsf{A_{T_i}}$, $i = 1, \ldots, n$.

Thus, an object-oriented dynamic system $\mathsf{DS(A_0)}$ of signature $D\Sigma$ consists of a set of instance algebras $|\mathsf{DS(A_0)}|$ called the *carrier* of $\mathsf{DS(A_0)}$ and a set

[1] See one of [1,2,3] for the definition of an update set and its use for producing a new state.

of observers, mutators, dependent functions and procedures. For example, the function associated with the mutator name *put_value* declared in the class $VecIt$ will produce an update set needed for updating the content of a location, and the function associated with the observer name *advance* will produce the identifier considered to be next to a given identifier in a given state. A call of the procedure *allocate* will eventually produce a new state by attaching a number of locations to the current state.

2.3 Terms and Their Interpretations

There are several rules for creating terms of object types following the conventions used in object-oriented languages. Moreover, a special kind of term called *transition term* is introduced to denote transitions from one algebra to another (transition rules are special cases of transition terms). The main rules for creating the terms are the following (we omit interpretation where it is self-evident).

1. If $at : T_1, \ldots, T_n \longrightarrow T'$ is an attribute/observer declaration from the object type signature marked with T, $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, and t is a term of type T, then $t.at(t_1, \ldots, t_n)$ is a term of type T'. Examples: $vit.value\_stored$, $vit.advance$, $vit.plus(2)$, $vit.less(vit1)$, where vit and $vit1$ are the names of dynamic constants of type $VecIt$. The term is interpreted by invoking the corresponding attribute/observer function in the current state.
2. If t is a term of type T, then $\mathbf{D}(t)$ is a term of type Boolean. **Interpretation**: $\mathbf{D}(t)^{\mathsf{A}} = \mathsf{true}^{\mathsf{A}}$ if t is defined in A, and $\mathbf{D}(t)^{\mathsf{A}} = \mathsf{false}^{\mathsf{A}}$ otherwise. The interpretation of this term allows us to check whether the argument term is defined in the current state. For example, the interpretation of $\mathbf{D}(vit.advance)$ will let us know whether there is an object identifier next to vit in the current state.
3. If $m : T_1, ..., T_n$ is a mutator declaration from the object type signature marked with T, $t_1, ..., t_n$ are terms of types $T_1, ..., T_n$, respectively, and t is a term of type T, then $t.m(t_1, ..., t_n)$ is a transition term called a *mutator call*. Example: $vit.put\_value(3)$. This kind of term serves for indicating an update of a mutable object. The term is interpreted by invoking the corresponding mutator.
4. If $f : T_1, \ldots, T_n \longrightarrow T$ is a dynamic/dependent function declaration and $t_1, \ldots, t_n$ are terms of types $T_1, \ldots, T_n$, respectively, then $f(t_1, \ldots, t_n)$ is a term of type T. The term is interpreted by invoking the corresponding dynamic/dependent function.
5. If $p : T_1, ..., T_n$ is a procedure declaration and $t_1, ..., t_n$ are terms of types $T_1, ..., T_n$, respectively, then $p(t_1, ..., t_n)$ is a transition term called a *procedure call*. The term is interpreted by invoking the corresponding procedure.

2.4 Axioms

Axioms accompanying the declarations of dependent functions are *data equations* of the form $t_1 == t_2$ where t_1 and t_2 are two terms of the same type. The

terms are composed of universally quantified variables, operation names from Σ_0, method names from Σ_{obj}, and function names from Σ_{din} and Σ_{dep}. The interpretation of such a term produces an algebra element. A data equation $t_1 == t_2$ is satisfied in a dynamic system iff the interpretation of both t_1 and t_2 produces the same algebra element in any its state. Since a term containing the name of a dynamic function can evaluate differently in different states, the function whose specification contains such a term generally produces different results in different states (i.e., the result depends on the state).

Axioms accompanying the declarations of procedures are *dynamic equations* of the form $t_1 == t_2$ where t_1 and t_2 are two transition terms. A transition term is either a procedure call or a mutator call or a transition rule. The interpretation of either of them produces an update set. In a dynamic equation $t_1 == t_2$, the first term is normally a procedure call or mutator call and the second one is a transition rule. A dynamic equation is satisfied in a dynamic system iff the interpretation of both terms in any its state produces the same update set.

Transition rules are conventionally created like in traditional ASMs [2] with an additional possibility of using procedure calls and mutator calls in rule constructors. There is, however, an important difference in the treatment of the assignment of an undefined value to a location. There can be no single *undef* value for all data types. Therefore, all dynamic functions are partial, and in an update rule

$f(t_1, ..., t_n) :=$ *undef*

undef is just a keyword indicating that $f(t_1, ..., t_n)$ becomes undefined.

We can now write the following axioms for the above VecIt signature (typical axioms for equality and non equality are omitted):

{**forall** i, i1: VecIt, x, n: Nat. – *declaration of universally quantified variables*
i.put_value(x) == i.value_stored := x; – *dynamic axiom*
i.get_value == i.value_stored; – *this one and all the others are data axioms*
i.advance == nat_to_id(id_to_nat(i) + 1);
i.retreat == nat_to_id(id_to_nat(i) – 1);
i.plus(n) == nat_to_id(id_to_nat(i) + n);
i.minus(n) == nat_to_id(id_to_nat(i) – n);
i.difference(i1) == id_to_nat(i) – id_to_nat(i1)
i.less(i1) == id_to_nat(i) < id_to_nat(i1);
i.greater(i1) == id_to_nat(i) > id_to_nat(i1);
i.leq(i1) == id_to_nat(i) ≤ id_to_nat(i1);
i.geq(i1) == id_to_nat(i) ≥ id_to_nat(i1)}

3 Generic Object Types

We cannot be satisfied with only concrete object types because many object types can have a similar structure and it would be tiresome to specify them again and again. Therefore a notion of generic object type is introduced. We start with the simplest case, *unconstrained generic object types.*

An *object-structured specification* Σ is defined above as a triple $\langle OTYPE, O\Phi, int^o\rangle$, where int^o is a function mapping $OTYPE$ into $O\Phi$. We propose the following way of constructing the elements of $OTYPE$, using two non intersecting sets of names, $\underline{T}$ and $\underline{R}$, called *names of concrete object types* and *names of generic object types*, respectively: if $T \in \underline{T}$, then $T \in OTYPE$; if $T_1, ..., T_n$ are elements of $\underline{T}$ and $R \in \underline{R}$, then $R(T_1, ..., T_n) \in OTYPE$. The elements of $OTYPE$ are called *type terms* in the sequel.

Let now $int^o(R(T_{11}, ..., T_{1n})) = Spec1$ and $int^o(R(T_{21}, ..., T_{2n})) = Spec2$, where $R(T_{11}, ..., T_{1n})$ and $R(T_{21}, ..., t_{2n})$ are type terms and $Spec1$ and $Spec2$ are object type specifications. We say that the object type $R(T_{11}, ..., T_{1n})$ is a *sibling* of the object type $R(T_{21}, ..., T_{2n})$ if the replacement of each T_{1i} with T_{2i}, $i = 1, ..., n$, in $Spec1$ converts it in $Spec2$. Thus, we can say that the object types $Vector(Integer)$ and $Vector(Char)$ are siblings if the replacement of $Integer$ with $Char$ in the specification of $Vector(Integer)$ converts it in the specification of $Vector(Char)$ and vice virsa. We can propose a special way of constructing a part of the function int^o for a family of object type siblings.

Let $q_1, ..., q_n$ be names (of type parameters). A pair $\langle R(q_1, ..., q_n), Spec\rangle$, where $R \in \underline{R}$ and $Spec$ is an object type specification additionally using $q_1, ..., q_n$ as type terms in method declarations, is part of the function int^o, such that for any type name $T_i \in \underline{T}$, $i = 1, ..., n$, $int^o(R(T_1, ..., T_n)) = Spec[T_1/q_1, ..., T_n/q_n]$, where $Spec[T_1/q_1, ..., T_n/q_n]$ is an object type specification produced by replacing each q_i in $Spec$ with T_i.

A pair $\langle R(q_1, ..., q_n), Spec\rangle$ is called a *generic type specification*. The replacement of type parameters with type names in both parts of a generic type specification is called a *generic type instantiation*. Note that due to the use of the function int^o, we do not need to introduce a special semantics for generic object types. A generic type specification in this approach is just a way of defining a part of this function. This corresponds one to one to the practice of modern programming languages (like C++) regarding generic object types as templates.

Two type terms $R(T_{11}, ..., T_{1n})$ and $R(T_{21}, ..., T_{2n})$ are equivalent if T_{1i} and T_{2i}, $i = 1, ..., n$, are the same type name.

Example. Taking into account that in the real memory we would like to have locations storing values of different types, we specify a generic vector iterator type:

class VecIt(T) = **spec**
[**attribute** value_stored: T;
mutator put_value: T;
observer get_value: T;
advance, retreat: VecIt(T);
plus, minus: Nat ⟶ VecIt(T);
difference: VecIt(T) ⟶ Nat;
eq, neq, less, greater, leq, geq: VecIt(T) ⟶ Boolean]
{**forall** x: T, i, i1: VecIt(T), n: Nat. – *declaration of universally quantified variables*
i.put_value(x) == i.value_stored := x; – *dynamic axiom*
i.get_value == i.value_stored; – *this one and all the others are data axioms*

i.advance == nat_to_id(id_to_nat(i) + 1);
i.retreat == nat_to_id(id_to_nat(i) – 1);
i.plus(n) == nat_to_id(id_to_nat(i) + n);
i.minus(n) == nat_to_id(id_to_nat(i) – n);
i.difference(i1) == id_to_nat(i) – id_to_nat(i1)
i.less(i1) == id_to_nat(i) < id_to_nat(i1);
i.greater(i1) == id_to_nat(i) > id_to_nat(i1);
i.leq(i1) == id_to_nat(i) ≤ id_to_nat(i1);
i.geq(i1) == id_to_nat(i) ≥ id_to_nat(i1)}.

4 Generic Vector Type

We can now give a formal specification of the generic vector type informally defined in [4]. It is assumed that vector components (which are vector iterators) are numbered starting from zero. The allocated number of the vector components is defined by the attribute *max_size*, the current number of initialized components is defined by the attribute *size*. Vector components in the range $0..size-1$ are called vector elements in the sequel. The clause **dom** in the specification indicates the domain of a partial function: in a domain specification **dom** $t : b$, the term t is defined if and only if b evaluates to *true*. A transition rule of the form **set** $R_1, ..., R_n$ **end** indicates the parallel execution of the transition terms $R_1, ..., R_n$.

class Vector(T) = **spec**
[**attribute** comp: Nat ⟶ VecIt(T); – *vector components, allocated by an allocator*
size: Nat; – *the current size of a vector, initialized by an allocator*
max_size: Nat; – *the maximum size of a vector, initialized by an allocator*
mutator empty_vec; – *default constructor*
initialized_vec: Nat, T; – *initialization of the content of the first n components*
copy: Vector(T); – *copy constructor*
push_back: T; – *append an element at the end of a vector*
pop_back; – *delete the last element of a vector*
insert: VecIt(T), T; – *insert an element at the position indicated*
erase: VecIt(T); – *remove the element indicated*
swap: Vector(T); - *swap the contents of two vectors*
observer empty: Boolean; – *is a vector empty?*
_[_]: Nat ⟶ T; – *fetch an element of a vector*
front, back: T; – *the first and last elements of a vector*
begin, end: VecIt(T)] – *the starting and terminating iterators of a vector*
{**forall** x: T, iv: VecIt(T), n: Nat, v, v1: Vector(T).
dom v[n]: n ≥ 0 & n ≤ v.size;
dom v.initialized_vec (n, x): n ≤ v.max_size;
dom v.copy(v1): v.max_size ≥ v1.size;
dom v.push_back(x): v.size < v.max_size;
dom v.insert(iv, x): v.size < v.max_size & iv.geq(v.begin) & iv.less(v.end);
dom v.erase(iv): iv.geq(v.begin) & iv.less(v.end) & v.size > 0;

```
v.empty_vec == v.size := 0;
v.initialized_vec (n, x) == set v.size := n,
    forall i: 0 .. n–1. v.comp(i).value_stored := x end;
v.copy(v1) == set v.size := v1.size,
    forall i: 0 .. v1.size–1. v.comp(i).value_stored := v1.comp(i).value_stored end;
v.push_back(x) == set v.comp(v.size).value_stored := x, v.size := v.size + 1 end;
v.pop_back == v.size := v.size – 1;
v.insert(iv, x) == set v.size := v.size + 1,
    forall i: 0 .. v.size-1. if v.comp(i).geq(iv)
        then v.comp(i+1).value_stored := v.comp(i).value_stored,
    iv.value_stored := x end;
v.erase(iv) == set v.size := v.size – 1,
    forall i: 0 .. v.size-1. if v.comp(i).geq(iv)
    then v.comp(i).value_stored := v.comp(i+1).value_stored end;
v.empty == v.size =0;
v[n] == v.comp(n).value_stored;
v.begin := v.comp(0);
v.end == v.begin.plus(v.size);
v.front == v.comp(0).value_stored;
v.back == v.comp(v.size-1).value_stored }.
```

5 Generic List Type

A list is a double-linked sequence of elements ordered according to the use of the constructor operations "push_back" (inserts an element at the end of a list), "push_front" (inserts an element at the beginning of a list), and "insert" (inserts an element in the middle of a list). All list elements are numbered starting with one for the first element. The number of the last element is equal to the number of list elements. A list element is an object of the corresponding iterator type ListIt. We define it first.

```
class ListIt(T) = spec
 [attribute value_stored: T;
     pred, next: ListIt(T);
 mutator put_value: T;
 observer get_value: T;
     advance, retreat: ListIt(T);
     eq, neq: ListIt(T) ⟶ Boolean]
 {forall i: ListIt(T), x: T.
  i.put_value(x) == i.value_stored := x;
  i.get_value == i.value_stored;
  i.advance == i.next;
  i.retreat == i.pred}.
```

Note that, in contrast to the object type VecIt, this object type does not possess the methods *plus* and *minus* since list elements are accessed only sequentially.

Now we can define the generic list type. Due to space limitations, the specification of some mutators is left to the reader. Note that a transition rule of the form **seq** $R_1, ..., R_n$ **end** indicates the sequential execution of the transition terms $R_1, ..., R_n$, and the transition rule of the form **while** b **do** R **end** indicates the repetition of the execution of the transition term R. The formal semantics of these rules can be found in [3].

class List(T) = **spec**
[**attribute** begin, end: ListIt(T); – *the starting and terminating iterators of a list*
size: Nat; – *current size of a list*
mutator empty_list; – *empty list constructor*
initialized_list: Nat, T; – *initialized list constructor*
copy: List(T); – *copy constructor*
push_front: T; – *insert an element at the beginning of a list*
push_back: T; – *insert an element at the end of a list*
pop_front; – *delete the first element of a list*
pop_back; – *delete the last element of a list*
insert1: ListIt(T), T; – *insert one element at the position indicated*
insertN: ListIt(T), Nat, T; – *insert N elements at the position indicated*
erase1: ListIt(T); – *remove the element indicated*
eraseN: ListIt(T), ListIt(T); – *remove the elements between the iterators*
observer empty: Boolean; – *is a list empty?*
front, back: T; – *first and last elements of a list*
has: ListIt(T) ⟶ Boolean; – *an auxiliary operation checking the presence*
– *of an element in the list*
precedes: ListIt(T), ListIt(T) ⟶ Boolean] – *an auxiliary operation verifying*
– *that the first iterator precedes the second one in a list*
{**forall** x: T, n: Nat, l, l1: List(T), i, i1: ListIt(T).
dom l.pop_front: l.size > 0; **dom** l.pop_back: l.size > 0;
dom l.insert1(i, x): l.size > 0 & ¬l.has(i);
dom l.insertN(i, n, x): l.size > 0 & ¬l.has(i);
dom l.eraseN(i, i1): l.size > 0 & l.has(i) & l.has(i1) & l.precedes(i, i1);
dom l.front: l.size > 0; **dom** l.back: l.size > 0;
l.empty_list == **import** new_elem: ListIt(T) **in**
set l.begin := new_elem, l.end := new_elem, l.size := 0 **end** ;
– *an iterator beyond the list component iterators is created; according to*
- *STL, it is the terminating list iterator*
l.push_back(x) == **import** new_elem: ListIt(T) **in**
set new_elem.pred := l.end, l.end.next := new_elem, l.end.value_stored := x,
l.end := new_elem, l.size := l.size + 1 **end** ;
l.initialized_list(n, x) == **seq** l.empty_list, **while** l.size < n **do** l.push_back(x) **end**;
l.copy(l1) == **seq** *sequence of transition rules* **end**;
l.push_front(x) == **import** new_elem: ListIt(T) **in** *set of transition rules*;
l.pop_front == **set** l.begin := l.begin.next, l.begin.next.pred := undef,
size := size – 1 **end**;
l.pop_back == *set of transition rules*;

l.insert1(i, x) == **import** new: ListIt(T) **in set** i.next := new, i.next.pred := new,
new.pred := i, new.next := i.next, new.value_stored := x, l.size := l.size + 1 **end**;
l.insertN(i, n, x) == **seq** l.insert1(i, x), **if** n > 1 **then** l.insertN(i.next, n–1, x) **end** ;
l.erase1(i) == **if** l.size = 1 **then** l.pop_front – *first element is deleted*
else set i.pred.next := i.next, i.next.pred := i.pred, l.size := l.size – 1 **end** ;
l.eraseN(i, i1) == **set** l.erase1(i), **if** i.next.neq(i1) **then** l.eraseN(i.next, i1) **end** ;
l.front == l.begin.value_stored; l.back == l.end.pred.value_stored;
l.has(i) == *Boolean term evaluating to "true" if l has i*;
l.precede(i, i1) == *Boolean term evaluating to "true" if i precedes i1 in l*}.

6 Object Type Categories

The generic constructs defined above permit us to specify algorithms over data structures, such as vectors or lists, abstracted from the type of the component of the structure. One of the STL requirements is the ability to specify also algorithms abstracted from the data structure itself, i.e. to specify an algorithm capable, for example, to manipulate both vectors and lists. *Object type categories* serve this purpose.

An object type category resembles a *sort class* of Spectrum [5] or *type class* of Ruslan [6] both based on *type classes* introduced in [8] and *sypes* introduced in [9]. It defines a set of object type specifications with some common properties.

Let CAT be a set of names, CS a set of specifications constructed like object type specifications with the use of an extra object type name "@", and $int^c : CAT \longrightarrow CS$ a function mapping names in CAT to specifications in CS. If $C \in CAT$, $cs \in CS$, and $cs = int^c(C)$, then the maplet $\langle C \mapsto cs \rangle$ is the specification of the object type category C.

Let $\langle C \mapsto cs \rangle$ be the specification of an object type category and $\langle T \mapsto ots \rangle$ the specification of an object type. It is said that the object type T is a type of the category C (or T belongs to C) if $cs[T/@] \in ots$, where $cs[T/@]$ is the specification cs with the symbol "@" replaced with T. That is, an object type belonging to a certain type category must include its specification as a subspecification. **Example**:

classcat Equal = **spec** – *category of object types with an equality operation*
[**observer** eq, neq: @ ⟶ Boolean]
{**forall** x, y: @, **exist** z: @.
x.eq(x) == true; x.eq(y) == y.eq(x);
x.eq(z) & z.eq(y) == x.eq(y);
x.neq(y) == ¬x.eq(y)}.

Any object type possessing the methods *eq* and *neq* specified as above is the type of the category Equal. We also consider that $TYPE$ is the name of the category with the empty specification and, therefore, any data/object type belongs to this category.

Constructing a type category, we can inherit the specification of an existing type category producing the union of the specifications as result.

Like an object type, an object type category can be generic, i.e., it can use type parameters in its specification. The definition of a generic object type category is the same as the definition of a generic object type. A generic object type belongs to a generic type category if their parameters match and it includes the specification of the type category as its subspecification. Iterator categories serve as examples.

1. Each iterator type of the following category has methods *advance* and *get_value* in addition to the methods of the category "Equal".[2]

classcat InputIterator(T) = **spec** Equal
[**observer** advance: @; – *produces the succeeding iterator*
get_value: T] – *reads the value stored in an iterator*

2. Each iterator type of the following category has methods *advance* and *put_value* in addition to the methods of the category "Equal".[3]

classcat OutputIterator(T) **spec** Equal
[**observer** advance: @; – *produces the succeeding iterator*
mutator put_value: T] – *stores a new value in an iterator*

3. Each iterator type of the following category has a mutator *put_value* in addition to the methods of the category "InputIterator".

classcat ForwardIterator(T) = **spec** InputIterator(T)
[**mutator** put_value: T]
{**forall** i: @, x: T.
i.put_value(x) == i.value_stored := x;
i.get_value == i.value_stored}

4. Each iterator type of the following category has the method *retreat* in addition to the methods of the category "ForwardIterator".

classcat BidirectionalIterator(T) = **spec** ForwardIterator(T)
[**observer** retreat: @] - *produces the preceding iterator*

5. Each iterator type of the following category has several methods in addition to the methods of the category "BidirectionalIterator".

classcat RandomAccessIterator(T) = **spec** BidirectionalIterator(T)
[**observer** plus, minus: Nat ⟶ @;
difference: @ ⟶ Nat;
less, greater, leq, geq: @ ⟶ Boolean]

[2] The category of input iterators is introduced in STL to allow iteration over input streams in the same way as, say, over vectors.
[3] The category of output iterators is introduced in STL to allow iteration over output streams in the same way as, say, over vectors.

According to the definitions, an object type VecIt(T) belongs to the type categories RandomAccessIterator(T), BidirectionalIterator(T), ForwardIterator(T), OutputIterator(T), InputIterator(T), and Equal. An object type ListIt(T) belongs to the type categories BidirectionalIterator(T), ForwardIterator(T), OutputIterator(T), InputIterator(T), and Equal (it does not belong to the type category RandomAccessIterator(T), however). Thus, a vector iterator can be used in any algorithm requiring either a random access iterator or bidirectional iterator or forward iterator or input iterator or output iterator. In the same way a list iterator can be used in any algorithm except that one which requires a random access iterator.

7 Constrained Genericity

According to the definitions of generic components in Section 3, any type can be used as instantiation argument. At the same time, it is often needed that only a type belonging to a certain type category could be substituted. Therefore, the definitions of generic components should be changed according to this constraint. The constrained generic object type is defined as follows.

Let $q_1 : C_1, ..., q_k : C_k$ be names (of type parameters) indexed with type category names. A pair $\langle R(q_1 : C_1, ..., q_k : C_k), Spec\rangle$, where $R \in \underline{R}$ and $Spec$ is an object type specification additionally using $q_1, ..., q_k$ as type terms in method declarations and operators from $C_1, ..., C_k$ in axioms, is part of the function int^o, such that for any type term T_i of type category C_i, $i = 1, ..., k$, $int^o(R(T_1, ..., T_k)) = Spec[T_1/q_1, ..., T_k/q_k]$, where $Spec[T_1/q_1, ..., T_k/q_k]$ is an object type specification produced by replacing each q_i in $Spec$ with T_i.

The constrained generic type category is defined in a similar way.

A *generic function declaration* is a triple $\langle gf, (q_1 : C_1, ..., q_k : C_k), FP^q\rangle$, where gf is a (generic) function name, $q_1 : C_1, ..., q_k : C_k$ are names (of type parameters) indexed with type category names and FP^q is a function profile additionally using $q_1, ..., q_k$ as type terms.

If $gf : \langle(q_1 : C_1, ..., q_k : C_k), FP^q\rangle$ is a generic function declaration and $T_1, ..., T_k$ are type terms of type categories $C_1, ..., C_k$, respectively, then $gf(T_1, ..., T_k) : FP$ is an *instantiated function declaration*, where FP is a function profile produced by replacing each q_i in FP^q with T_i; $gf(T_1, ..., T_k)$ is called an *instantiated function name*. Instantiated function names are used for producing data terms in the same way as ordinary function names are used.

If $gf : \langle(q_1 : C_1, ..., q_k : C_k), FP^q\rangle$ is the declaration of a dynamic function, then an algebra A of the given signature is provided with a function $\mathsf{map}^{\mathsf{A}}_{\mathsf{gf}}$ binding an instantiated function name $gf(T_1, ..., T_k)$ to a function as it is described in Section 2.2 for dynamic functions.

If $gf : \langle(q_1 : C_1, ..., q_k : C_k), FP^q\rangle$ is the declaration of a dependant function, then a dynamic system $\mathsf{DS}(\mathsf{A}_0)$ is provided with a function $\mathsf{map}^{\mathsf{DS}(\mathsf{A}_0)}_{\mathsf{gf}}$ binding an instantiated function name $gf(T_1, ..., T_k)$ to a function as it is described in Section 2.2 for dependant functions.

A generic function specification consists of a generic function declaration and a set of axioms using instantiated function names in their terms.

Notation: we use the brackets **gen** ... **profile** to embrace type parameters in a generic function declaration.

Examples:

1. Specify a function which looks for a value in a data structure between *first* and *last* iterators and returns the iterator storing the value if it is found and the last iterator if the value is not found.

```
function find: gen I: InputIterator, T: TYPE profile: I(T), I(T), T ⟶ I(T);
{forall first, last: I(T), value: T.
 find(I, T)(first, last, value) ==
  if first.get_value = value ∨ first.eq(last) then first
  else find(I, T)(first.advance, last, value)}.
```

Now, if we have the following declarations:

```
vec: Vector(Integer);
list: List(Char);
```

we can invoke the function in the folowing ways:

```
find(VecIt, Nat)(vec.begin, vec.end, 7);
find(ListIt, Char)(list.begin, list.end, 'a');
```

In this case both vector iterators and list iterators can be used because both belong to the category of input iterators required in the function specification. Note the substitution of the generic types $VecIt$ and $ListIt$ for the generic type category I. In the next example list iterators cannot be used.

2. Specify a function which performs binary search in a structure containing ordered components and returns the iterator containing the element to be found (for simplicity we assume that the structure really contains the element).

```
function binary_find: gen I: RandomAccessIterator, T: Ordered
                                  profile I(T), I(T), T ⟶ I(T);
{forall first, last: I(T), value: T.
 binary_find(I, T)(first, last, value) == let d = last.difference(first), h = d/2,
                                  current = first.plus(h), cv = current.get_value in
  if cv = value then current
  elseif value < cv then binary_find(I, T)(first, current, value)
  else binary_find(I, T)(current, last, value)}.
```

Now, we can call the function with vector iterators like the following:

```
binary_find(VectIt, Nat)(vec.begin, vec.end, 7)
```

because vector iterators belong to the class RandomAccessIterator, and we cannot call it with list iterators. Note the use of the type category Ordered (not defined here) which is needed to make sure that the type of the components contains the operation "<".

The above examples show us in which way the functions abstracting both from the structure and the type of its components can be specified.

Generic procedures are declared similarly to generic functions and are specified similarly to ordinary procedures. The vector allocator, a special procedure which allocates memory for a particular vector (according to STL, such a procedure is associated with each container class), can serve as an example. The import rule of Gurevich ASMs [2] is used in the specification for creating a new object identifier.

proc vec_allocator: **gen** T: TYPE **profile** Vector(T), Nat;
{**forall** v: Vector(T), n: Nat.
vec_allocator(T)(v, n) ==
set v.max_size := n, v.size := 0,
forall i = 0 .. n-1. **import** new_elem: Vector(T) **in** v.comp(i) := new_elem
end – *vector attributes are initialized.*

If *intvec* is a dynamic constant of type $Vector(Integer)$, then *vec_allocator(Integer)(intvec, 100)* will allocate 100 noninitialized locations to *intvec* and set initial values to its attributes max_size and $size$.

8 Related Work

We are not going to discuss here the approaches representing object states as elements of the same algebra. The work along this approach is heavily based on traditional algebraic specification methods. We can only repeat after F. Parisi-Presicce and A. Pierantonio that "the algebraic framework so far has been inadequate in describing the dynamic properties of objects and their state transformation as well as more complex notions typical of the object oriented paradigm such as object identity and persistency of objects" [10]. The interested reader can refer to [11,12,13,14].

We review here several works considering object states as algebras. Object-oriented extensions of the prominent specification methods VDM [15] and Z [16] are the first of them.

VDM++ [17] introduces a notion of class definition as a template for a collection of objects possessing a number of instance variables (internal state) and methods (external protocol). The definitions of the methods of existing classes can be inherited when a new class is defined (multiple inheritance). Object's initial state and invariant can be specified. A set of statements typical of imperative programming languages is provided. Unfortunately, the description of the semantics of the language is done rather informally, in the way the semantics of programming languages is usually described. As a result, the user gets an impression that VDM++ is a programming language provided with some specification facilities (pre- and post-conditions) rather than a specification language. No genericity is provided in the language.

Object-Z [18] practically has the same features as VDM++ with the difference that it is based on Z. A class is here a set of attributes and a set of operations acting upon these attributes. In contrast to VDM++, the semantics

of Object-Z is formally given. The state is considered as a function from a set of identifiers (attributes) to the set of all possible values. One can say that it corresponds to a homogeneous algebra whose signature contains only constant symbols (compare it with a more general case of ASM where functions as algebra components play a significant role). A class definition can be supplied with a number of type parameters, a kind of unconstrained genericity is provided in this way. No notion of class category and, respectively, constrained genericity exists in the language. Object creation is also not provided by the language. Therefore a specification similar to the specification of list types given above is not possible.

Z++ [19,20] is another development based on Z providing facilities comparable to those of Object-Z. The main difference is that its syntax is quite different from that of Object-Z (which actually follows the syntax of Z) and is chosen to stress the commonalty of the language with object-oriented programming languages like Eiffel. Formal semantics of the language is defined by means of category theory. However, the main attention is paid to the formal definition of refinements between classes while such important notions as state, class, object, etc. are considered well-known and not defined in the semantics. Like in Object-Z, a class definition in Z++ can be supplied with a number of type parameters providing unconstrained genericity. Again, no notion of class category (in our sense) and, respectively, constrained genericity exists in the language. The semantics of a generic class specification is also not reported.

In [21] objects are elements of instance structures which are quadruples of algebras of different signatures. Specifications of the algebras resemble traditional algebraic specifications. One of the algebras is extended with extra "state function symbols" mapping object identifiers to their values. Dynamic operations serving for object evolution are modeled by algebra morphisms. The author believes that the specification of these operation should have an imperative nature, but he does not suggest a method of specification. The approach is further formalized with a heavy use of category theory in [10]. In contrast to all of this, we represent the state by a single algebra, we believe that there is no necessity for state function symbols since user-defined observers perfectly serve for this purpose, and we suggest a concrete method of specification by means of transition rules.

The "Hidden Sorted Algebra" approach [11], where some sorts are distinguished as hidden and some other as visible, treats states as values of hidden sorts. Visible sorts are used to represent values which can be observed in a given state. States are explicitly described in the specification in contrast to our approach. This work combined with Meseguer's rewriting logic [22] has served as the basis of the dynamic aspects of the language CafeOBJ [23]. In this language states and transitions are modeled, respectively, as objects and arrows belonging to the same rewrite model which is a categorical extension of the algebraic structure. Meseguer's rewriting logic is also the basis of the specification language Maude [24].

The specification language Troll [25] should be mentioned as one of the main practical achievements in the field. Troll is oriented to the specification of objects where a method (event) is specified by means of evaluation rules similar to equations on attributes. Although the semantics of Troll is given rather informally, there is a strong mathematical foundation of its dialect Troll-light [26] with the use of data algebras, attribute algebras and event algebras. A relation constructed on two sets of attribute algebras and a set of event algebra, called *object community*, formalizes transitions from one attribute algebra to another. Although Troll possesses generic facilities, non of them is formalized in [26].

The problem of constrained genericity in object-oriented class libraries is discussed in [27]. Abstract classes resembling our object type categories are used there for constraining the generic parameters, and a notion of syntactical conformance is introduced for checking whether a concrete class is an instance of a given abstract class. There is an important difference between an abstract class and a concrete class since one cannot create an object of an abstract class. In fact, an abstract class specifies some methods common for a community of concrete classes. For this reason me make a clear difference between object types and object type categories independently specified. An additional feature of our approach is generic object type categories standing for communities of generic object types.

Finally, a fundamental work [28] formalizing bounded parametric polymorphism similar to our constrained genericity should be paid attention. Here genericity is constrained by allowing only those type arguments which are subtypes of the parameter type. In this respect, this approach is very similar to the previous one. Another peculiarity of the work is that an object does not possess a unique identifier, it is just a tuple of methods, and object updates are simulated by method overrides generally producing new objects.

9 Conclusion

The mechanisms for the specification of generic object types and type categories are introduced in the paper. With the use of these mechanisms, many generic algorithms abstracting from the type of the data structure being manipulated can be easily specified. Although the general technique of generic type specification is well-known and can be found in the literature, the novelty of this work is the extension of the technique to the case of object-oriented ASMs. Another novelty is the definition and use of generic type categories for the specification of generic algorithms.

The described technique has permitted us to specify some components of the Standard Template Library for C++. The library thus specified can be easily adapted to another object-oriented language. The experience obtained in the process of the specification has proved the power of the technique. Its main features can be summarized as follows:

1. We represent immutable values by data types and specify them algebraically.

2. We represent mutable objects possessing states by object types and specify them by means of transition rules.

3. We define generic (data, object) types to abstract from the type of the component.

4. We define (data, object) type categories to abstract from the structure.

5. We define a generic algorithm by means of transition rules manipulating the objects thus specified.

Tools supporting this style of specification remain the subject of further work.

Acknowledgements. I thank anonymous referees for their helpful comments.

References

1. Y. Gurevich. *Evolving Algebras 1993: Lipary Guide*. In: Börger, E., ed., Specification and Validation Methods, Oxford University Press, 1995, pp. 9-36.
2. Y. Gurevich. *May 1997 Draft of the ASM Guide*. University of Michigan EECS Departmental Technical Report CSE-TR-336-97 (available electronically from http://www.eecs.umich.edu/gasm/).
3. A.V. Zamulin. *Object-Oriented Abstract State Machines*. Proc. ASM workshop, Magderburg, Germany, 21-22 September, 1998, pp. 1-21 (available electronically from http://www.eecs.umich.edu/gasm/).
4. D.R. Musser and A. Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, 1996.
5. Broy, M., Facchi, C., Grosu, R., ea. *The Requirement and Design Specification Language Spectrum, An Informal Introduction, Version 1.0*. Technische Universitaet Muenchen, Institut fuer Informatik, April 1993.
6. A.V. Zamulin, *The Database Specification Language RUSLAN: Main Features*. East-West Database Workshop (proc. Second International East-West Database Workshop, Klagenfurt, Austria, September 25-28, 1994), Springer (Workshops in Computing), 1994, 315-327.
7. P.D. Mosses. *CoFI: The Common framework Initiative for Algebraic Specification and Development*. In: M. Bidoit and M. Dauchet, eds., TAPSOFT'97: Theory and Practice of Software Development, LNCS, vol. 1214, pp. 115-137.
8. P. Wadler and S. Blott. *How to make ad-hoc polymorphism less ad-hoc*. Conf. Record of the 16th ACM Annual Symp. on Principles of Progr. Lang., Austin, Texas, January 1989.
9. Nakajima, R., Honda, M., and Nakahara, H. *Hierarchical Program Specification: a Many-sorted Logical Approach*. Acta Informatica 14, pp. 135-155 (1980).
10. F. Parisi-Presicce and A. Pierantonio. *Dynamic behavior of Object Systems*. In: Recent trends in Data Type Specification. LNCS, vol. 906, 1995, pp. 406-419.
11. J.A. Goguen and R. Diaconescu. *Towards an Algebraic Semantics for the Object Paradigm*. Recent Trends in Data Type Specification, LNCS, 1994, vol. 785, pp. 1-29.
12. H.-D. Ehrig and A. Sernadas. *Local Specification of Distributed Families of Sequential Objects*. In: Recent Trends in Data Type Specifications. LNCS, vol. 906, 1994, pp. 219-235.
13. F. Parisi-Presicce and A. Pierantonio. *An Algebraic Theory of Class Specification*. ACM Transactions on Software Engineering and Methodology, April 1994, vol. 3, No. 2, pp. 166-169.

14. C. Cristea. *Coalgebra Semantics for Hidden Algebra: Parameterised objects and Inheritance*. in: Recent Trends in Algebraic development Techniques. LNCS, vol. 1374, 1997, pp. 174-189.
15. C. B. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.
16. J. M. Spivey. *understanding Z. A specification language and its formal semantics*. Cambridge University Press, 1988.
17. E.H. D,rr and J. van Katwijk. *A Formal Specification Language for Object Oriented Designs*. In: Computer Systems and Engineering (Proceedings of CompEuro'92). IEEE Compute Society Press, 1992, pp. 214-219.
18. R. Duke, P. King, G.A. Rose, and G. Smith. *The Object-Z specification language*. In: T. Korson, V. Vaishnavi, and B. Meyer, eds., Technology of Object-Oriented Languages and Systems: TOOLS 5, Prentice hall, 1991, pp. 465-483.
19. K. Lano and H. Houghton, eds., *Object-Oriented Specification case Studies*. Prentice Hall (object-oriented series), 1994.
20. K. Lano and H. Houghton. *The Z++ manual*. October 1994. Available from ftp://theory.doc.ic.ac.uk/theory/papers/Lano/z++.ps.Z
21. A. Pierantonio. *Making Statics Dynamic*. In: G. Hommel, editor, Proc. International Workshop on Communication based Systems, Kluwer Academic Publishers, 1995, pp. 19-34.
22. J. Meseguer. *Conditional rewriting logic as a unified model of concurrency*. Theoretical Computer Science, vol. 96, No 1 (April 1992), pp. 73–155.
23. R. Diaconescu and K. Futatsugi. *CafeOBJ Report*.World Scientific Publishing Co. Pte. Ltd, AMAST series in Computing", vol. 6,1998.
24. J. Meseguer. *A logical theory of concurrent objects and its realization in the Mode language*. In: Research Directions in Concurrent Object-Oriented Programming. The MIT Press, Cambridge, Mass., 1993, pp. 314-390.
25. T. Hartmann, G. Saake, R. Jungclaus, P. Hartel, and J. Kush. *Revised Version of the Modelling Language TROLL*. Technishe Universitaet Braunschweig, Informatik-Berichte 94-03, 1994.
26. M. Gogolla and R. Herzig. *An Algebraic Semantics for the Object Specification Language TROLL-light*. In: Recent Trends in Data Type Specifications, LNCS, vol. 906, 1995, pp. 290–306.
27. A. Frick, G. Goos, R. Newmann, W. Zimmermann. *Construction of Robust Class Hierarchies*. Software—Practice and Experience, 2000 (to be published).
28. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

Towards an ASM Thesis for Unconventional Algorithms

Wolfgang Reisig

Humboldt-Universität zu Berlin
reisig@informatik.hu-berlin.de

Abstract. All descriptions of algorithms, be they formal or informal, employ data structures, operations on them, and some policy to cause operations be applied to data.
Gurevich calls a formal description technique for algorithms *algorithm universal* if it allows for each informally described algorithm a formal representation that would essentially make precise the notions used in the informal description, not employing additional data, operations or steps.
Gurevich's *ASM thesis* claims Abstract State Machines be algorithm universal for conventional, sequential algorithms.
Here we are behind properties of formal presentations that are algorithm universal for unconventional, distributed algorithms.

1 Conventional and Unconventional Algorithms

1.1 Algorithmic Ideas

The design of a computer embedded solution to a real world problem always includes phases of *algorithmic ideas*. An algorithmic idea describes intuitive, tentative, not yet fully specified aspects of an intended algorithm. Though inevitably vague, an algorithmic idea is intended to be transformed into an unambiguous, formally presented algorithm.

Experience reveals that all algorithmic ideas cover three kinds of components:

First, a choice of *passive* components. This includes data items in various roles (e.g. data stored in data bases, data in a transient buffer, or data amenable for processing), variables with their respective actual values, and potential location of control items.

Second, a choice of *active* components. This includes operations to observe or to change (update) passive components, e.g. reading a variable, writing a variable, establishing new control items, transporting data items to other places (as e.g. in case of input and output), providing new roles or contexts to data items.

Third, a *policy* to activate active components and to cause their action. This includes *sequential* policies that cause sequences of action occurences, as well as *parallel* policies that cause sequences of sets of action occurrences, and *distributed* policies, that cause actions to occur in partially unspecified order.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 112–130, 2000.

1.2 Conventional Algorithms

We denote an algorithm *conventional* if it sticks to the conventional paradigm of computing: Upon all input available, the algorithm starts. Elementary actions are executed in sequence, conditional alternative, and conditional iteration. Runs should terminate and provide output. Non-terminating runs are considered errornous, providing no output. Such an algorithm computes a partial function from the set of potential inputs to the set of potential outputs.

The intermediate choice of actions may not be unique, in which case one of them is selected nondeterministically. The algorithm computes a relation in this case.

1.3 Reactive Algorithms

As a first liberalization of the conventional paradigm, not all input is required to be initially available, nor is all output postponed until termination: Input and output may occur *during* a run, as outlined in Fig. 1.

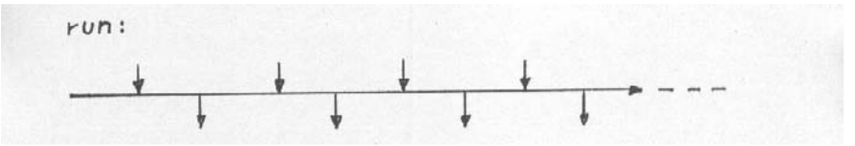


Fig. 1. Outline of a run of a reactive algorithm

This yields a number of consequences: Firstly, a run may *wait* for fresh input. 'Waiting' is a really new aspect, not present in any conventional theories of computation. Secondly, a run of such an algorithm is sensible also if it diverges. Different diverging runs may execute different interactions with their environment.

Each run of this kind of algorithm transforms a finite or infinite sequence of input data into a finite or infinite sequence of output data: The algorithm *reacts* to each new input item with fresh output items; hence the term of *reactive* algorithms.

Typical examples of such algorithms are technical control algorithms, such as e.g. lift control.

1.4 Cooperating Algorithms

Two reactive algorithms A and B may cooperate, with output of A used as input for B and vice versa, output of B used as input for A. In addition, the two algorithms may cooperate with their environment. Fig.2 outlines this situation. Typical examples of such algorithms are communication protocols.

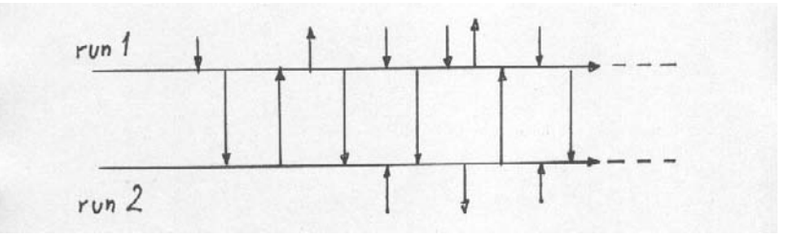


Fig. 2. Outline of a run of cooperating algorithms

1.5 Network Algorithms

The idea of cooperating algorithms can be generalized to *networks* of local algorithms. Each local algorithm in such a network is directly linked to its neighbor algorithms, as outlined in Fig.3.

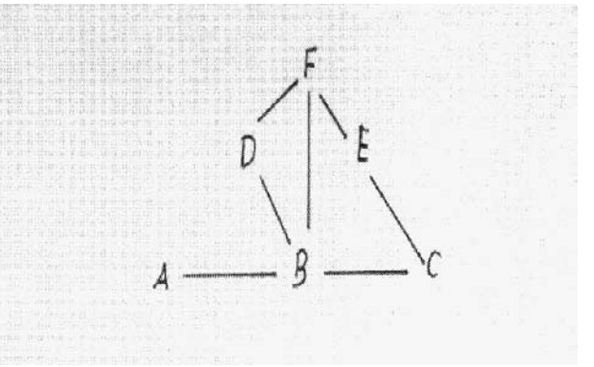


Fig. 3. Network of local algorithms $A, \ldots, E$

They may perform a task in cooperation, sending messages to neighbors, as sketched in Fig.4.

1.6 Shared Memory and Threads on-the-Flay

A *shared memory algorithm* consists of two or more threads of actions, performed on storage private to the threads, as well as on common storage. Threads of control may be generated as well as destroyed during an algorithm's run.

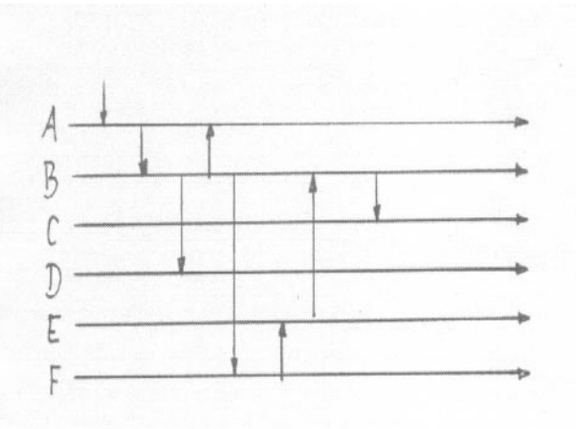


Fig. 4. Outline of a run of the network in Fig.3

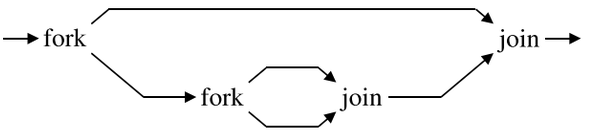


Fig. 5. Regularly structured threads of control

Fig. 5 shows a regularly structured split of thread of control.In contrast, not all parts of thread of control can uniquely be assigned to a pair of fork and join in Fig.6.

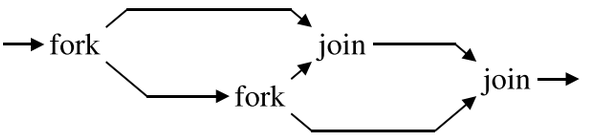


Fig. 6. Irregularly structured threads of control

Fork and join may dynamically be generated during a run of an algorithm. The overall number of parallel threads thus is not fixed or predictable beforehand. The notion of determined thread of control fades away.

1.7 Liberal Policies of Action

Some versions of unconventional algorithms do with quite liberal policies of action. As an - admittedly extreme - example, consider the set of symbols $\{a, b, c, d, e\}$ with the standard alphabetic order, $<$, and the set $\{1, \ldots, 5\}$ of indices with their natural order. Furthermore, assume an initial pairing of letters and indices, e.g. $(a, 5)$, $(b, 1)$, $(c, 3)$, $(d, 2)$, $(e, 4)$. Our task is to order the symbols properly, resulting in the pairs $(a, 1), \ldots (e, 5)$. The only available operation, swap $((x, i), (y, j))$, is defined for letters x, y and indices i, j. This operation is activated if $x > y$ and $i < j$, in which case it returns (x, j) and (y, i).

The crucial point is the policy of action. We describe it intuitively by conceiving each pair (x, i) as a *fish*, x, with a *sticker*, i.

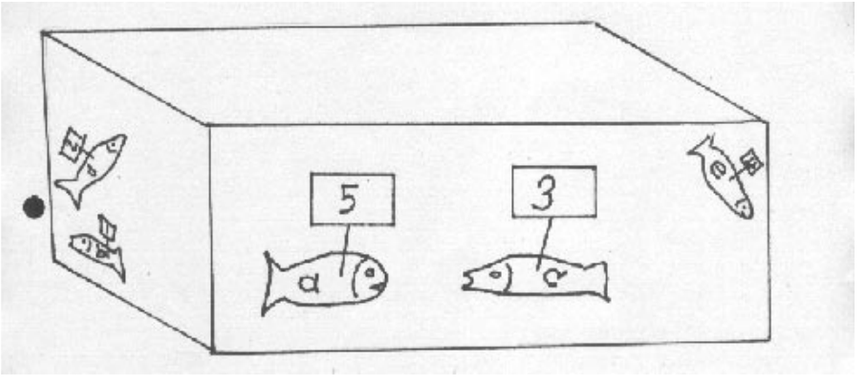


Fig. 7. Outline of distributed sorting

The fishes are swimming around in an aquarium, as sketched in Fig.7. Whenever two fish meet, they exchange their stickers in case the order of fishes and stickers do not coincide.

Is this an algorithm at all? What minimal requirements guarantee termination? How detect termination?

1.8 Progress and Quiescence

Conventional algorithms are always to *proceed*. For example, execution of a PASCAL program does not stall with an enabled statement: all statements of PASCAL are *progressing*. Unconventional algorithms may have actions that are not progressing, but *quiescent*: they may remain enabled forever.

As an example, any reasonable algorithm that organizes mutual exclusion (mutex) must do with at least three states for each involved site. Calling those states *quiet*, *pending*, and *critical*, a *quiet* site must spontaneously be able to

become *pending*, but must also be able to remain *quiet* forever; the step from *quiet* to *pending* is "quiescent". The algorithm furthermore must guarantee that *pending* will eventually lead to *critical* and back to *quiet*. Hence all other actions are "progressing".

Algorithms are frequently composed with other algorithms along quiescent actions. Quiescence then hints at additional preconditions that are not explicitely stated in the algorithm itself.

1.9 Fairness Assumptions

Nondeterministic choice among two or more actions can be required to be performed *fairly*: a run is discarded, if it infinitely often is to decide the same alternative and decides always in favor of the same action (up to finitely many exceptions).

Fairness assumptions are irrevelant for conventional algorithms. Fairness assumptions leave the scope of computable functions; they can be verified or falsified only with infinite behavior. Fairness can not be implemented literally, one usually implements a stronger assumption, ruling out more runs.

As an example, all mutex algorithms need a fairness assumption [11]. This assumption is frequently shifted into fair reading of variables. Then the debate of forthcoming Chapter 2 applies: If fair reading of shared variables is guaranteed, the problem is solved.

1.10 Schemata for Algorithms

Algorithms frequently don't determine all aspects of computation. The most prominent parameter for computation is *input*. Unconventional algorithms do with further parameters.

For example, the *Echo algorithm* is a schema to broadcast acknowledged messages within connected, undirected networks of agents and message lines. Algorithmic behavior, concurrent runs etc. manifests in *single* networks only. The Echo algorithm is just a *schema* for many algorithms: each concrete graph defines a specific algorithm.

1.11 Specification of Algorithms

A *specification* of an algorithm does not formulate operational behavior, but just states its crucial properties. For example, each mutex algorithm assumes two partners, l and r, with states *quiet*, *pending* and *critical*, and the steps as discussed in 1.8. In addition, never both partners are coincidently *critical*. Any algorithm that meets those requirements, is a perfect mutex algorithm.

More involved specifications do with requirements that guarantee distinguished behavior only if they can rely on proper input.

1.12 Algorithm Universal Formalisms

A formal presentation may stick more or less tightly to an algorithmic idea. The most liberal fashion would suggest that any algorithmic idea essentially describes a computable function. A formal presentation of the algorithmic idea then formalizes this function by any mathematical means. The tightmost fashion was a formal presentation that would give the passive and active components of the algorithmic idea a precise meaning, avoiding any new components, and likewise would reflect the algorithmic ideas' activation policy in such a way that the steps of the formal presentation bijectively correspond to the steps of the algorithmic idea. Gurevich in [8] calls such a formal description technique for algorithms *algorithm universal*.

The *sequential ASM thesis* of [8] claims that sequential ASMs are algorithm universal for conventional algorithms, as described in 1.1, as well as for reactive algorithms as described in 1.2.

Here we are behind the problem of universal description techniques for all unconventional algorithms, some of which we outlined above.

We will not commit ourselves as to whether or not parallel and distributed ASMs would meet this property. Instead, in addition to the above described issues, in the rest of the paper we discuss some aspects of unconventional algorithms that must be respected by any universal description technique.

The above classification takes up an ongoing discussion on algorithmic concepts that would not just compute partial functions. [16] made reactive algorithms popular. For cooperating and Network algorithms, cf. [15], [1] and [20]. Shared memory and dynamically generated threads have been studied for decades cf. [4]. Liberal Policies of actions and schemata are covered in [20], among others. Specification issues are mainly discussed in the framework of Temporal Logic, such as e.g. in Unity [CM88] or TLA [14]. Fundamental aspects of liberal policies of algorithms are discussed in [17] and [21].

2 The Issue of Reading

It may come as a surprise that we conceive reading the actual value of variables the first issue to be considered here. Reading is a quite obvious, simple operation in conventional algorithms.

Unconventional algorithms, however, frequently consist of local agents that share variables: A shared variable can be addressed by two or more agents. If they do so coincidently, the outcome is debatable. There are two main approaches to tackle this problem.

2.1 Reading Doesn't Matter

The idea of this approach is one distinguished agent who *controls* the variable and may both read (test) and write (change) it. All other agents are only allowed to read. Reading is assumed not to really interfere with writing. Lamport in [13]

nicely exemplifies this approach by some kind of a flag that can be risen and lowered by one person only. Others may observe the scenery.

The observer (reader) of the flag does not affect the flag in any respect. In particular, the observer of the flag can not prevent the flag's owner from rising and lowering.

2.2 Reading Matters

Lamport's above described approach properly reflects our everyday physical environment. It includes myriads of light waves that are reflected by items' surface, e.g. flags, and later on catched by humans eyes. This is why observation does not affect the observed.

Not all physical universes are structured like this. An example is the universe of computer hardware.

Reading a variable, i.e. copying the actual value of a register, is physically realized as a sequence of actions that indeed affect the register's contents. This contents may be unreliable, inpredictable or in fact different from the expected value at intermediate states. While being read, the register can not be written at the same time.

A register like this behaves like a notebook, used by a group of agents. Both actions of reading and writing require exclusive control. It is the agents' *access* to the notebook that counts for the purpose of writing and reading alike.

2.3 Conclusion

To what extent do the above considerations affect the construction of algorithms? Conventional algorithms are not affected at all, but unconventional algorithms are affected to a great extent. For example, a lot of algorithms that organize mutual exclusion are sensible only in the framework of the first approach, where reading is assumed not to affect a variable.

There is a traditional bias to conceive the assumptions of the first approach more convenient and natural than the second. This is why operating systems frequently offer a surface to processes that pretend the assumptions of the first approach, in particular interference free reading. The second approach will however turn out more lucid for a systematic approach to unconventional algorithms.

3 Sequential and Concurrent Runs

3.1 The Notion of Runs

The notion of a single run is fundamental for algorithms. In fact, an algorithm's primarly purpose is to characterize infinitely many (and, frequently, infinitely long) runs.

A run comprizes *occurences of actions*, called *events*. Events can be ordered according to different aspects. One aspect assumes a (totally ordered) time scale, defining a time stamp for each event. A run then is *lock step* if two or more events may be given the same time stamp. Otherwise it is *sequential*.

Another aspect describes *causal dependency*. Any event causes preconditions for its *direct successor events*. Transitive closure of the direct successor relation then is a partial order which describes the *causal dependencies* among events. Runs with this order are *concurrent*. Long standing disputes emphasize advantages and disadvantages of sequential and concurrent runs.

Sequential runs are frequently considered more natural, convenient and widespread. In the context of algorithm universal formalisms however, concurrent runs are inevitable. Forthcoming sections have the details.

3.2 An Example

As an example consider Dijkstra's well known system of five philosophers $a, \ldots, e$, sitting around a table. Each philosopher alternates the actions of *thinking* and *eating*. As an additional requirement, neighbored philosophers are never to eat together. Each pair of neighbors shares a *fork* to this end. A philosopher can start eating only if he finds his adjacent two forks *available*. In this case he picks them up (in which moment they are no longer available) and makes them available again only upon terminating *eating*.

ASMs suggest to freely choose a level of abstraction at our convenience. So we choose action A to comprize philosopher a picking up his adjacent forks, eating, and releasing his forks. Actions B, C, D and E are defined accordingly.

3.3 Sequential Runs

What are the sequential runs of the above algorithm? Apparently, each infinite sequence $w \in \{A, \ldots, E\}^\infty$! On the given level of abstraction, neighbored philosophers, never eating together, can not be distinguished from detached philosophers who very well might eat together.
Writing $\overset{x}{y}$ to denote concurrent events x and y, one may construct lockstep runs

$$\ldots \quad \begin{matrix} x_1 & x_2 \\ y_1 & y_2 \end{matrix} \quad \ldots$$

where x_i and y_i denote events of detached philosophers ($i = 1, 2, \dots$). This works nicely for four philosophers. However, lockstep runs can not adequately represent, for example, an occurence of A while an occurrence of C is followed by an occurence of D.

3.4 Refinement

To overcome the shortcomings of sequential and lockstep runs, one may *refine* the actions $A, \dots, E$. For example, A may be replaced by three actions A_1, A_2 and A_3, denoting the philosopher a to pick up his adjacent forks, to eat and to release his forks, respectively. A sequence beginning $A_1\ A_2\ B_1$ was then ruled out, for obvious reasons.

Interleaved and lockstep runs consisting of refined actions could more truly express causal relations than the above runs of unrefined actions can. But ASMs insist in the specifiers right to choose an abstraction level at his convenience. A proper way to solve the problem was the use of *concurrent runs*, to be considered next.

3.5 Concurrent Runs

We introduce *concurrent runs* of the philosophers algorithm on the above introduced level of actions $A, \dots, E$. Those runs very well allow to distinguish the requirement of neighbors not eating concurrently. Graphically, let

represent the above described action A, with ingoing arcs denoting the forks of A being available to A, and the outgoing arcs denoting the forks of A being released by A.

Fig.8 then shows one (out of infinitely many) concurrent runs of the philosophers' algorithms: Philosophers a and c start concurrently, followed by b twice in a row. d acts after c, and e after a and d.

3.6 Concurrent Determinism

Concurrent runs nicely depict a whole bunch of important aspects of behavior. *Concurrent determinism* is one of them: An algorithm is *concurrently deterministic* if each initial state yields *exactly one* concurrent run. As an example assume the above described system of five philosophers, with the additional requirement of *decent behavior*: Neighbored philosophers alternate use of their shared fork.

Any fixed initial appointment of each fork to one of its users then yields a unique concurrent run. Fig.9 and Fig.10 exemplify two such decent runs. Fig.11 outlines different patterns in those runs.

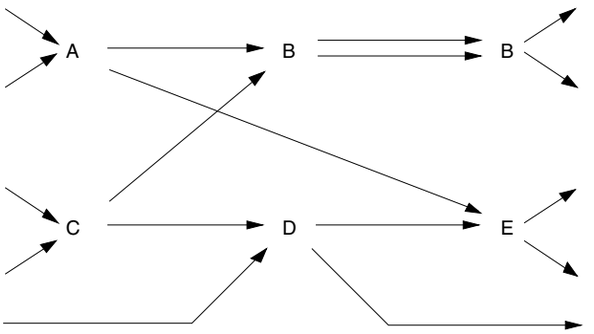


Fig. 8. A concurrent run of the philosophers algorithm

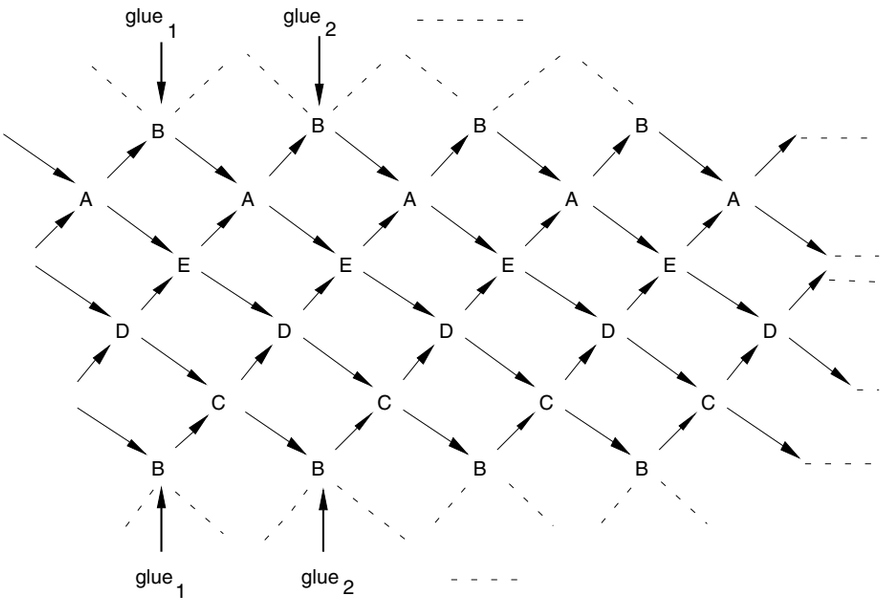


Fig. 9. A decent run of the philosophers system

3.7 Local States in Concurrent Runs

The above examples of concurrent runs consist of occurences of five actions, $A, \ldots, E$. Direct succession of two such occurences, graphically depicted by e.g. $C \to D$ in Fig.9 implicitely includes a *local state* that describes the common fork of C and D as being released by C, apt to be picked up by D.

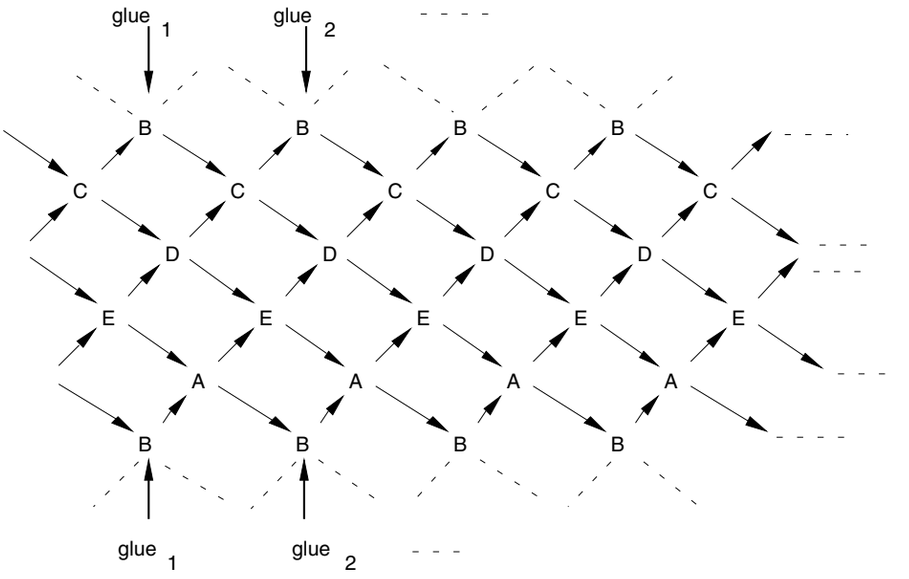


Fig. 10. A further decent run of the philosophers system

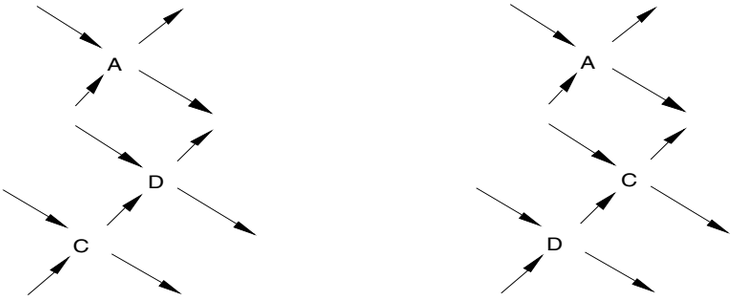
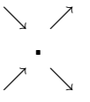


Fig. 11. Different patterns in the runs of Fig.9 and Fig.10

Local states in concurrent runs are frequently much more prominent: There are concurrent runs that are essentially characterized by their local states. The runs of the sorting algorithm of Chapter 1.7 illustrate this.

With xi denoting the pair (x, i) consisting of 'fish x with sticker i', Fig. 12 and Fig. 13 show two runs of the sorting algorithm.

An event is represented as



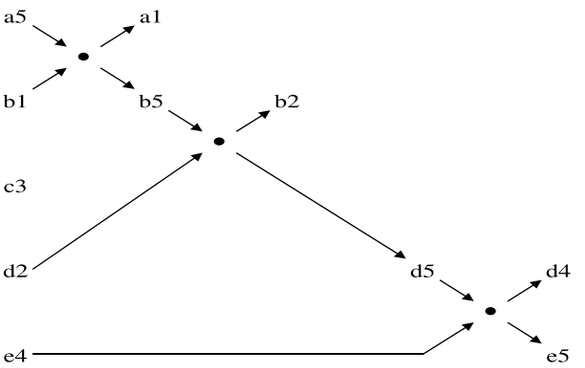


Fig. 12. A concurrent run of distributed sorting

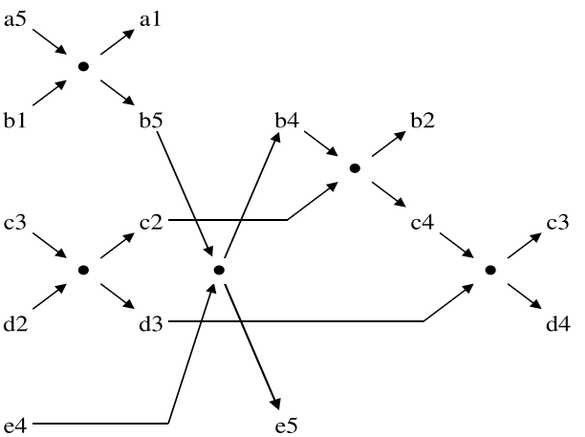


Fig. 13. A further concurrent run of distributed sorting

Each event is an instance of the action of two fishes swapping their stickers. The involved fishes and stickers are indicated by the local states presented at the arcs starting and ending nodes, respectively.

3.8 Is There Really More in Concurrent Runs?

The sequential and concurrent runs of an algorithm are tightly interwined: A concurrent run, i.e. a partially ordered set of events, uniquely defines a set of total extension of its order. This yields sequences of actions. Each such sequence of course is a sequential run of the algorithm. The total extensions of all concurrent runs are just the sequential runs.

Vice versa, can the concurrent runs be derived from the sequential runs? This depends on knowledge about the involved actions' *scope*, the topic of the next chapter.

3.9 A Short History of Concurrent Runs

Research into concurrent runs started with [10] as well as [9] , suggesting acyclic graphs with nodes denoting events and arcs denoting causal precedence. Petri in [18] describes how to construct the concurrent runs of condition - event systems, the most elementary class of Petri nets. Lamport advocates concurrent runs in [12] , but renounces them in [14]. Concurrent runs have been suggested for several versions of Petri Nets [5] , [3] , and other system models [7]. They are extensively employed in [20]. The issue has been debated in an electronic discussion on the concurrency mailing list, reprinted in [19].

4 The Scope of Actions

The *scope* of an action includes all variables that the action addresses. This notion is not too important for conventional algorithms. It is however decisive when unconventional algorithms are to be specified, constructed and analyzed.

4.1 An Example

As an example consider two integer variables `x` and `y`, and a Boolean variable, `z`. Let

$$\mathit{init} \equiv \quad \texttt{x} := 0; \quad \texttt{z} := \texttt{true}; \quad \texttt{y} := 0$$

Furthermore, let

$$\begin{aligned} X \equiv &\quad \texttt{while true do} \quad x := x + 1; \\ Y \equiv &\quad \texttt{while true do} \quad y := y + 1; \end{aligned}$$

$$\mathit{program} \quad \equiv \quad \mathit{init}\ ; \quad X \parallel Y$$

The meaning of the program is obvious: initialization *init* is followed by parallel execution of X and Y. With X and Y *atomic* actions, the sequential runs of *program* are all infinite words $w \in \{X, Y\}^\infty$ with both X and Y occuring infinitely often in w.

program has exactly one concurrent run, to be depicted as in Fig.14

$$\begin{array}{ll} \mathtt{x} = 0 & X \to X \to X \to \cdots \\ & \mathtt{z} = \mathtt{true} \\ \mathtt{y} = 0 & Y \to Y \to Y \to \cdots \end{array}$$

Fig. 14. A concurrent run of *program*

4.2 A Variant

As a variant of the above program, assume the variables $\mathtt{x}, \mathtt{y}, \mathtt{z}$ and the initialization *init* as above.
Furthermore, let

$$\begin{array}{lll} \overline{X} \equiv & \mathtt{while\ z\ do} & \mathtt{x} := \mathtt{x} + 1; \\ \overline{Y} \equiv & \mathtt{while\ z\ do} & \mathtt{y} := \mathtt{y} + 1; \end{array}$$

and

$$\overline{program} \equiv \quad init; \quad \overline{X} \parallel \overline{Y}$$

The sequential runs of *program* and $\overline{program}$ coincide.

The concurrent runs of $\overline{program}$ depend on the approach to reading of variable $\mathtt{z}$, as discussed in Chapter 2. Concurrent runs are based on the assumption that reading affects the variable, as described in 2.2. $\overline{program}$ then exihibits infinitely many concurrent runs. One of them can be depicted as in Fig.15.

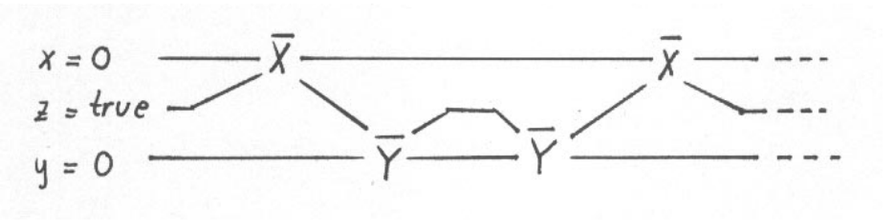


Fig. 15. A concurrent run of $\overline{program}$

4.3 Void Arguments

As a further example for the crucial role of the scope of actions, let x and y be variables over some domain, A, and let $f : A \to A$ be a function. The semantics of $x := f(x)$, in a concurrent run can be depicted as in Fig. 16.

In this figure, f denotes an event that transforms the local state of x, actually $x = n$, into the local state $x = f(n)$, thereby not touching the local state of y, given by $y = m$.

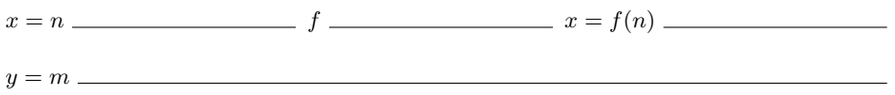


Fig. 16. Occurrence of $x := f(x)$

Now, let $g : A \times A \to A$ be defined by $g(a, a') = f(a)$, i.e. the second argument of g does not affect the outcome of g. Hence, $f(x)$ and $g(x, y)$ yield the same result for all values of x and y. Replacing $f(x)$ by $g(x, y)$ does not change any result of a sequential run.

The scope of $f(x)$ is $\{x\}$, whereas the scope of $g(x, y)$ is $\{x, y\}$. The semantics of $x := g\{x, y\}$ in a concurrent run is depicted in Fig. 17.

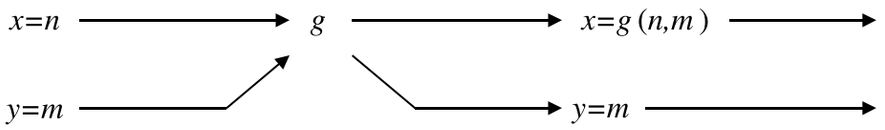


Fig. 17. Occurrence of $x := g\,(x, y\,)$

In this figure, g denotes an event that transforms the local states of both x and y, actually $x = n$ and $g = m$, respectively, into the local states $x = f(n)$ and $y = m$.

Summing up, $f(x)$ and $g(x, y)$ would cause equal effects in sequential runs, but different effects in concurrent runs.

4.4 Deriving Concurrent Runs from Sequential Runs

Summing up the above observation, concurrent runs explicitely respect, represent and exploit the scope of actions, whereas sequential runs don't. This yields different notions of *equivalence*: The two programs of 4.1 and 4.2 as well as the above assignment statements $x := f\{x\}$ and $x := g\{x, y\}$ may be mutually substituted in sequential runs. They may however not be substituted in concurrent runs, because they would change the actions' scope.

We are now prepared to return to the question stated in 3.7 and state the proposition:

> *The concurrent runs of an algorithm can be derived from its sequential runs if and only if the involved actions' scope is known.*

5 Proper Data Structures

5.1 Inconsistent Update

A typical example of inconsistent update was an ASM - sequence (or 'block') formed

$$f(a) := b$$
$$f(a) := c$$

with $b \neq c$. [8] requires computation to stall in this case. [7] suggests nondeterministic choice.

Both alternatives assume a runtime system or a global controller or whatever means to detect and to manage this case instantly. Distributed algorithms prevent this kind of regime, however: If $f(a) := b$ and $f(a) := c$ are concurrently executed by two different agents, there is no means to detect inconsistency instantly.

5.2 How Manage Distributed Inconsistency?

Static analysis of an algorithm may reveal that distributed inconsistency never occurs. But in case it does occur, it can not be detected instantly upon occurrence. Hence it can neither be prevented nor treated in a particular way. So we must provide a framework that would allow distributed inconsistency to occur.

An obvious choice for such a framework were *relations*, i.e. sets of pairs $R \subseteq A \times B$, for sets A and B. A function $f : A \to B$ is then conceived as the relation $\hat{f} := \{(a, f(a)) \,|\, a \in A\}$. Distributed updates $f(a) := b$ and $f(a) := c$ would then extend $\hat{f}$ by (a, b) and (a, c).

The idea of relations is not yet satisfactory. As an example, consider three agents A, B, C, concurrently executing $f(a) := b, f(a) := c$ and $f(a) :=$ *undef*, respectively. This would not yield a determined effect: It was now necessary to explicitely state the element to be removed, i.e. to replace $f(a) :=$ *undef* by *remove* (a, b) *from* f or *remove* (a, c) *from* f.

Still, a difficulty remains. It can be exemplified by the example of three agents, A, B, C, concurrently executing $f(a) := b, f(a) := b$ and *remove* (a, b) *from* f, respectively. This, again, yields no determined effect. There is a solution, however: With $f(a) := b$, A inserts one item (a, b) to f. With the same assignment statement, B inserts another item (a, b) to f. With *remove* (a, b) *from* f, C removes one of those items. Removing both of them would require another occurence of *remove* (a, b) *from* f.

Summing up, a data structure can concurrently be updated if it allows for *multiple* occurrences of items; the structure is to be a *multiset* over a set M. That multisets are the adequate basis for datastructures of distributed system, is well known for decades: Multiple occurences of identical tokens is a basis notion of Petri Nets. Multisets are likewise used in the Chemical Abstract Machine of [2].

The set M is frequently a relation or a graph of a function; it may be ordered, be equipped with other operations or distinguished elements. It may be a matter of fact for a given algorithm, that no element occurs at different locations at the same time. Nevertheless, a framework is required to capture the case of multiple occurrences. To this end we suggest two universal operations for all multisets over some set A, and $a \in A$:

insert (f, a)
is always enabled and inserts another item a to f;

remove (f, a)
is enabled if there exists at least one item a in f; occurrence of *remove* (f, a) then removes a from f.

Both these operations may occur concurrently. A nice illustration was a large buffet table with several waiters inserting plates of food and many clients, removing the plates from the buffet table.

The buffet paradigm does not imply, however, that the most general case of multisets always applies. The structure of an algorithm may guarantee that a dynamically changing data structure always meets distinguished properties. Typically, items may actually occur at most once; they may consist in pairs of items that form a tree, or are right unique (i.e. form a partial function), etc. Even total functions can be gained by help of an operator *update* (f, a, b, c) that consists of *remove* $(f, (a, b))$ and *insert* $(f, (a, c))$.

References

1. Hagit Attiya and Jenifer Welch. *Distributed Computing*. McGraw Hill, 1998.
2. Jean-Pierre Banâtre and Daniel Le Métayer. Programming by multiset transformation. *CACM*, 36(1):98–111, 1993.
3. Eike Best and Cesar Fernandez. *Nonsequential Processes*. Springer-Verlag, 1988.
4. W. P. de Roever et al. Concurrency verification: Introduction to compositional and noncompositional proof methods. to appear 2000.
5. U. Goltz and W. Reisig. The non-sequential behaviour of Petri nets. *Information and Control*, 57(2-3):125–147, 1983.
6. Yuri Gurevich. Evolving algebras–a tutorial introduction. *Bulletin of the EATCS*, (43):264–284, 1991.
7. Yuri Gurevich. Evolving algebras 1993 : Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
8. Yuri Gurevich. The sequential asm thesis. *Bulletin of the EATCS 67*, pages 93–124, 1999.
9. Anatol Holt. Introduction to occurrence systems. In *Associative Information Techniques*, pages 175–203. American Elsevir, New York, 1971.
10. Anatol Holt, H. Saint, R. Shapiro, and S. Warshall. Final report on the information systems theory project. Technical Report RADC-TR-68-305, Rome Air Developement Center, Griffis Air Force Base, New York, 1968. Distributed by Clearinghouse for Scientific and Technical Information, US Department of Commerce, 352 pages.
11. Ekkart Kindler and Rolf Walter. Mutex needs fairness. *Information Processing Letters 62*, pages 31–39, 1997.
12. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *CACM*, 21(7):558–565, 1978.

13. Leslie Lamport. Solved problems, unsolved problems and non-problems in concurrency. In *Proceedings of the* 3^{rd} *Symposium on Principles of Distributed Computing, 1983*, pages 34–44, 1984.
14. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
15. Nancy Lynch. *Distributed Algorithms.* Kaufman, Morgan, Los Altos, CA, 1996.
16. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, Berlin, 1992.
17. Robin Milner. Elements of interaction. *CACM*, 36:78–89, 1993.
18. C.A. Petri. Non-sequential processes. Technical Report Internal Report, GMD-ISF-77-5, Gesellschaft für Mathematik und Datenverarbeitung, Bonn(Germany), 1977.
19. Vaugham Pratt (ed.). Debate '90: An electronic discussion on true concurrency. In Peled, Pratt, and Holzmann, editors, *Partial Order Methods in Verification*, volume 29 of *DIMACS Series*, pages 359–403. 1997.
20. Wolfgang Reisig. *Elements of Distributed Algorithms.* Springer-Verlag, 1998.
21. Peter Wegner. Interactive foundations of computing. *Theoretical Computer Science*, 192:315–351, 1998.

Partially Ordered Runs: A Case Study

Yuri Gurevich[1] and Dean Rosenzweig[2]

Microsoft Research, USA
gurevich@microsoft.com and University of Zagreb, Croatia
dean@math.hr

Abstract. We look at some sources of insecurity and difficulty in reasoning about partially ordered runs of distributed ASMs, and propose some techniques to facilitate such reasoning. As a case study, we prove in detail correctness and deadlock–freedom for general partially ordered runs of distributed ASM models of Lamport's Bakery Algorithm.

Introduction

Distributed ASMs [4] is a general concurrent model of multi–agent computation. It was intended, in generalization of its more limited precursors [5,3], to allow as much concurrency as logically possible. Although the definition has been in print for several years, its notion of partially ordered runs has remained largely unexploited in its generality—most of its uses in the literature have recoursed to some kind of specialization to linear time, discrete or continuous. The general partially ordered runs seem to be somehow difficult to handle and to reason about [1].

Apart from deeply engrained intuitions of linear time, we feel that some rather technical sources of this difficulty can be detected.

Sequential runs [4] have two properties which greatly facilitate reasoning.

1. Every move is executed in a well–defined state.
2. 'External' (or 'monitored', cf. below) changes can be located in time, and thought of as actions by the environment. The environment thus becomes just another (typically implicit) agent, whose behaviour can be specified in declarative terms. The judicious splitting of dynamics to a part given by the program and a part that can be specified declaratively is a natural way to separate concerns and to abstract in ASMs.

In a partially ordered run neither of the above properties hold in general—a (global) state in which a move is executed is in general not uniquely defined, and it is not at all clear how to locate external changes in a partial order. These seem to be important sources of insecurity and difficulty in reasoning about partially ordered runs.

[1] This remark is not limited to the ASM context—most formal methods modelling concurrency tend to fall back, one way or another, to some kind of interleaving, sequential semantics

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 131–150, 2000.

We address both issues here, by developing some techniques to salvage as much of properties 1. and 2. as needed for partially ordered runs. In order to convince the reader that, with such means, nontrivial reasoning about partially ordered runs is feasible, we do it: as a case study, we apply the techniques to a nontrivial correctness proof. The case study also demonstrates that the proper setting for analysis of concurrent algorithms involves truly concurrent runs—mapping to linear time may well miss some important points.

In section 2 we deduce some simple consequences of the coherence condition of [4], providing sufficient conditions for moves in a distributed run to have at least a significant portion of state well–defined when they execute. This largely reconstructs property 1. for partially ordered runs of many distributed programs.

In the same section we introduce 'external change' in form of 'monitored' moves, by unknown agents with unknown programs, located exactly in the partial order. This is an extension of the standard practice (of having 'the environment' as a single, typically implicit, unknown agent) reconstructing largely property 2. for partially ordered runs.

In the rest of the paper we explain a nontrivial correctness proof for partially ordered runs: we prove in detail correctness and deadlock–freedom of (distributed ASM models of) Lamport's Bakery Algorithm [6]. We proceed on three different abstraction levels there. The abstraction level of our primary model, $\mathcal{B}_1$, corresponds precisely to that of Lamport's algorithm—the part specified declaratively is exactly what Lamport's algorithm doesn't define. A higher level description, of the model $\mathcal{B}_2$, allows us to deduce correctness and deadlock–freedom from abstract properties, shown to hold of $\mathcal{B}_1$. A lower level description, of the model $\mathcal{B}_0$, is an ASM shown to implement programatically exactly all behaviours allowed by $\mathcal{B}_1$.

Egon Börger and we wrote about the Bakery Algoirithm earlier; see [2] where a correctness proof was given for ASM models of the Bakery Algorithm with runs embedded in continuous linear time. Our new models and proofs are similar to those of [2], but they are also different. The proofs of [2] rely essentially on continuous linear time. As we will see later (see section 7), certain information about partially ordered runs is obfuscated in linear runs. Here we remove the linear time crutches and work directly with partially ordered runs. As in [2], we borrow ideas from [6] and [1]. In order for the paper to be reasonably self–contained we spell the entire construction out in full.

1 Preliminaries

We presume that the reader is familiar with [4]. Consider a one-agent program π and let f be a basic function of π which is dynamic so that the values of f can change in runs of π. Egon Börger suggested to use for ASMs the following terminology borrowed from Parnas. f is *controlled* if only π can change it. f is *monitored* if only the environment can change it. f is *shared* if both π and the environment can change it. (In [4], controlled functions were called internal, and monitored functions were called external.)

The terminology extends naturally to a multi-agent program Π. Let X be a set of agents of Π. A dynamic basic function f is controlled by X if only agents in X can change it. f is monitored by X if none of the agents in X can change it. The terminology also extends to particular locations rather than whole functions.

2 Partially Ordered Runs

We rely on the notion of partially ordered run of [4].

This means that we shall consider partially ordered sets of moves with the 'finite history' property: $\{y : y < x\}$ is finite for all x (we shall refer to the ordering relation by $<$, using also $\leq, >, \geq$). Each move is performed by an agent and, since agents are sequential, moves by one agent form a sequence; since there are finitely many agents, all antichains are finite.

A (global) state $\sigma(I)$ is associated with every finite initial segment I (a downwards closed finite subset) of a run, resulting from performing all moves in I so that if $s < t$ then s is executed earlier than t. In particular, if I is the empty segment then $\sigma(I)$ is the initial state of the run.

Using a partial order implies that moves s, t may be concurrent, i.e. incomparable: neither $s \leq t$ nor $t \leq s$ holds. The global state 'resulting' from a move is then in general not uniquely determined. It depends on what global state we see as the one in which the move is performed. Thus states are subject to a coherence condition [4], which allows us to give the following definitions.

Let $\mathrm{Post}(t)$ be the set of all finite initial segments in which a move t is maximal, and let a be the agent performing t. For each $I \in \mathrm{Post}(t)$, the state $\sigma(I)$ is the state obtained when a executes its program in the state $\sigma(I \setminus \{t\})$

'The global state' in which a move is performed is also not in general uniquely defined. Let $\mathrm{Pre}(t) = \{I \setminus \{t\} : I \in \mathrm{Post}(t)\}$. Several different states are thus in general associated with $\mathrm{Pre}(t)$, which makes reasoning about partially ordered runs somewhat difficult. The coherence condition may however, as we shall see below, impose that some term has a unique value at all states $I \in \mathrm{Pre}(t)$.

In order to express such requirements succinctly, we extend term valuation from states to some statesets, saying that

$$\mathrm{Val}_{\mathrm{Pre}(t)}(u) = \begin{cases} c & \text{if } \forall I \in \mathrm{Pre}(t)\ (c = \mathrm{Val}_{\sigma(I)}(u)) \\ \text{undef} & \text{if no such value exists} \end{cases}$$

where t is a move in a distributed run, $\mathrm{Val}_S(u)$ is the value of term u in state S, [4]. We shall often shorten $\mathrm{Val}_{\mathrm{Pre}(t)}(u)$ to $u_{\mathrm{Pre}(t)}$. When the value of $u_{\mathrm{Pre}(t)}$ is given by the first clause, i.e. when there is a c such that $\forall I \in \mathrm{Pre}(t)\ (c = \mathrm{Val}_{\sigma(I)}(u))$, we shall say that $u_{\mathrm{Pre}(t)}$ is *indisputable* (or that its value is indisputable). Notice that an indisputable value may also be undef, but whenever $u_{\mathrm{Pre}(t)} \neq$ undef then its value is indisputable.

For future reference, let us note some immediate properties of $\mathrm{Pre}(t)$.

Fact 1. *Let t be a move in a partially ordered run. The set Pre(t) has the following properties.*

1. *Pre*(t) *has a minimal element* $\min Pre(t) = \{s : s < t\}$ *and a maximal element* $\max Pre(t) = \{s : s \not\geq t\}$.
2. *Pre*(t) *is the set of all initial segments* I *such that* $\min Pre(t) \subseteq I \subseteq \max Pre(t)$, *and is hence closed under unions and intersections.*
3. *Let* s *be another move. The following are equivalent:*
   (a) s *is concurrent with* t.
   (b) $\min Pre(s) \cup \min Pre(t)$ *is an initial segment that belongs to* $Pre(s) \cap Pre(t)$.
   (c) $Pre(s) \cap Pre(t) \neq \emptyset$.
   (d) *There exist* $I, J \in Pre(t)$ *with* $s \in I \setminus J$.

Statement 3 may need an argument. We prove that (a) implies (b) implies (c) implies (d) implies (a).

(a) implies (b). Assume that s is concurrent with t and let $I = \min \mathrm{Pre}(s) \cup \min \mathrm{Pre}(t)$. Clearly I is an initial segment. Check that $\min \mathrm{Pre}(t) \subseteq I \subseteq \max \mathrm{Pre}(t)$, so that $I \in \mathrm{Pre}(t)$. By symmetry, $I \in \mathrm{Pre}(s)$.

(b) implies (c). Trivial.

(c) implies (d). Assume that $\mathrm{Pre}(s) \cap \mathrm{Pre}(t) \neq \emptyset$ and let J be any member of $\mathrm{Pre}(s) \cap \mathrm{Pre}(t)$. Set $I = J \cup \{s\}$. Clearly, $I \in \mathrm{Pre}(t)$.

(d) implies (a). Suppose that $I, J \in \mathrm{Pre}(t)$ and $s \in I \setminus J$. If $s < t$ then $s \in \min \mathrm{Pre}(t) \subseteq J$ so that $s \in J$ which is impossible. The dual argument shows that $s > t$ is impossible as well.

We shall say that a move t (in a given partially ordered run) *may change* the value of term u if, for some $I \in \mathrm{Pre}(t)$, we have $\mathrm{Val}_{\sigma(I)}(u) \neq \mathrm{Val}_{\sigma(I \cup \{t\})}(u)$ (equivalently, if t changes the value of u in some linearization of the run). If the above holds for all $I \in \mathrm{Pre}(t)$ (equivalently, if t changes the value of u in all linearizations), we shall say that t *must change* the value of u.

Recall that a linearization of a partially ordered run is a run with the same moves, and a linear order extending the given partial order. It was noted in [4] that, in view of the coherence condition, all linearizations of a finite run have the same final state.

Example 1. Take for instance two agents a, b, such that a executes the program

```
x := 1
```

and b executes the program

```
if mode = first then
   mode := second
   y := 1
endif
if mode = second then
   y := max(x,y)+1
   mode := final
endif
```

Now assume that $x = y = 0$, mode = first initially and consider a run with a move s of a, concurrent with two consecutive moves t_1, t_2 of b. Then both s and t_1 may but not must change the value of $\max(x, y)$, while t_2 must change it.

In these terms, we have

Lemma 1. *If $u_{Pre(t)}$ is not indisputable, then there is a move s concurrent with t which may change u.*

Proof. Assume the conclusion is false, that no move concurent with t may change u. To go from $\sigma(\min \mathrm{Pre}(t))$ to $\sigma(I)$, by Fact 1, we have to execute only some moves concurrent to t, none of which may change u. Thus $\mathrm{Val}_{\sigma(I)}(u) = \mathrm{Val}_{\sigma(\min \mathrm{Pre}(t))}(u)$ for all $I \in \mathrm{Pre}(t)$, and $u_{\mathrm{Pre}(t)}$ is indisputable, in contradiction to the premise. □

Lemma 2. *If there is move s concurrent with t which must change u, then $u_{Pre(t)}$ is not indisputable.*

Proof. Since s, t are concurrent, by Fact 1 there is an initial segment $I \in \mathrm{Pre}(s) \cap \mathrm{Pre}(t)$. Then both I and $I \cup \{s\}$ are in $\mathrm{Pre}(t)$, and, since s must change u, they have different values of u. □

We shall say that a term u is *focused* (in a given run) if any move which may change its value, also must change it. For a focused term u it is unambiguous to say that a move changes the value of u. In many cases the property of being focused will be obvious from the programs. Here we will also use the following lemma.

Lemma 3. *If the value of a term may only be changed by a single agent, then the term is focused.*

Proof. Suppose that u may be changed only by agent a and that t is a move by a. It suffices to show that $u_{\mathrm{Pre}(t)}$ is indisputable, since then $u_{\mathrm{Post}(t)}$ is also indisputable, and is different from $u_{\mathrm{Pre}(t)}$ iff t changes u. To see that $u_{\mathrm{Pre}(t)}$ is indisputable, note that the agents involved in all moves concurrent to t are different from a, and by the premise none of them may change u. Then by lemma 1 $u_{\mathrm{Pre}(t)}$ is indisputable. □

Putting the above lemmata together, we have

Fact 2. *If term u is focused, then $u_{Pre(t)}$ is indisputable iff t compares, in the given partial order, with all moves changing the value of u.*

The above example shows that the assumption of focus cannot be dropped for one direction of fact 2: $\max(x, y)_{\mathrm{Pre}(t_2)}$ is indisputably 1, and t_2 is concurrent to s, which may change the value. The nice property that every location is changed by one agent only does not help with the term $\max(x, y)$. Note also that fact 2 is useful for 'counter' updates like $c := c + 1$; if c is changed only in such a way, even concurrently, then it is focused.

Fact 3. *If the values of $u_{Pre(t)}$ and $u_{Pre(s)}$ are both indisputable but different, then $s < t$ or $t < s$.*

Proof. Assume $u_{\mathrm{Pre}(t)}$ and $u_{\mathrm{Pre}(s)}$ are both indisputable, and s and t are concurrent. Then there is an initial segment $I \in \mathrm{Pre}(s) \cap \mathrm{Pre}(t)$, and $u_{\mathrm{Pre}(s)} = \mathrm{Val}_{\sigma(I)}(u) = u_{\mathrm{Pre}(t)}$. □

A novelty of this paper is in allowing a run of a multi-agent program Π to have 'monitored moves', that is moves of unknown agents. We have some known agents with known programs. The moves of these agents are by definition *controlled*.

In addition, there may be some number of unknown agents whose programs are unknown as well. Their moves are by definition *monitored*.

A dynamic function of Π, or a location, will be called controlled if it is controlled by the known agents.

It will be called monitored if it is monitored by the known agents.

The presence of unknown agents is consistent with [4], even though the standard practice has been so far to assume that all explicit moves belong to known agents with known programs, though the active environment could make some implicit moves.

The moves by the environment now become explicit, and the unique monitored agent of standard practice, 'the environment', is now allowed to split to a number of possibly different, unknown agents.

The total number of agents involved in any given state is still finite. The coherence condition applies to all moves, controlled or monitored (even though we may have no direct way to verify instances of the coherence condition that involve monitored moves). Therefore facts 2 and 3 remain valid.

The presence of monitored moves allows us to separate concerns and to abstract. Parts of the algorithm can be formulated in form of more or less declarative statements about monitored moves in runs (which blurs to an extent the distinction between the algorithm and its environment).

3 Lamport's Algorithm

For arbitrary but fixed N let $P_1, \ldots, P_N$ be processes (we shall also talk about 'customers') that may want from time to time to access a 'critical section' CS of code. Any mutual exclusion protocol—which each P_i is supposed to execute in order to enter the critical section—has to prevent two processes from being in the critical section simultaneously. The Bakery Algorithm provides each P_i with a (shared) register R_i and a (private) array $n[1], \ldots, n[N]$ holding natural numbers. Only P_i is allowed to write to R_i but every process can read the register. We assume each register to be initialized with value 0.

The algorithm was presented by Lamport with the following piece of pseudocode.

Start

```
n[i] := 1
write(R_i,n[i])
```

Doorway

```
for all j≠i, read(R_j,n[j])
```

Ticket

```
n[i] := 1 + max_j n[j]
write(R_i,n[i])
```

Wait

```
for all j≠i, repeat
   read(R_j,n[j]) until
   n[j]=0 or n[j]>n[i] or (n[j]=n[i] and j>i)
```

Critical Section

Finale

```
R_i := 0
```

The Bakery Algorithm is divided into six consecutive phases: *start*, *doorway*, *ticket* assignment, *wait* section, *critical section* and *finale*.

To declare its interest in accessing the critical session, a process P_i writes 1 into array variable n_i and then posts the written value in its register.

In the doorway section, P_i copies all the other registers into its array. It then computes a *ticket*, which is the least integer greater than all integers in its private array, writes the ticket into n_i and posts the written value in its register.

During the subsequent wait section, process P_i keeps reading, into its array, the registers of each other process P_j, until the resulting array value $n[j] = 0$ or $n[j] > n[i]$ or $n[j] = n[i] \wedge j > i$.

The meaning of the condition is the following: if $n[j] = 0$, then P_j is not interested in entering the critical section, and it has no right to block P_i. If $n[j] > n[i] > 0$, then P_i has a smaller 'ticket' and has the right to go before P_j. The last clause resolves the case of two customers obtaining the same 'ticket': then one with smaller identifier goes first. Note that by ordering pairs of positive integers lexicographically:

$$(i, j) < (k, l) \longleftrightarrow [i < k \text{ or } (i = k \text{ and } j < l)]$$

one can write the until condition as follows: n[j]=0 or (n[j],j)>(n[i],i).

Once permitted to go, P_i enters the critical section. Upon leaving CS, as finale, P_i sets its register to 0.

Note also that the for-all commands in the doorway and the wait section may be executed in many ways, in various sequences, all at once, concurrently etc.

It may be worth mentioning the following. The process first writes into $n[i]$ and then posts the written value at R_i. Obviously it could do the two actions in the reverse order. Intuitively, the order between the two actions is immaterial, but the sequential character of the pseudo-code imposes one.

4 The Primary Model: ASM $\mathcal{B}_1$

The doorway section in Lamport's program does not give us any indication how customer i is supposed to perform reading. Should it read the registers R_j in the order given by the indices, in the reversed order? Should it get help and use vassal agents, one per each R_j ? There are many other possibilities. To reflect the situation in proper generality, our primary ASM model $\mathcal{B}_1$ includes no reading instructions whatsoever. Instead, we will require that runs of $\mathcal{B}_1$ satisfy certain provisos that guarantee that reading is performed.

4.1 The Program of $\mathcal{B}_1$

The ASM has only one program, used by all customers, which has five rules. The array $A(X, Y)$ represents the array $n[Y]$ of the program, private to customer X. We assume that initially all registers have value 0, all customers are in mode satisfied, and all elements of the array $A(X, Y)$ are undef. We assume that the identifiers of the N customers are distinct natural numbers $< N$. Variables X, Y will range over customers.

Start

```
if mode(me) = satisfied then
   A(me,me) := 1, R(me) := 1, mode(me) := doorway
```

Ticket

```
if mode(me) = doorway and ∀Y ≠ me (A(me,Y) ≠ undef) then
   A(me,me) := 1 + max_Y A(me,Y), R(me) := 1 + max_Y A(me,Y)
   mode(me) := wait
```

Entry

```
if mode(me) = wait and
∀ Y ≠ me (A(me,Y)=0 or (A(me,Y),id(Y)) > (A(me,me),id(me))) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   R(me) := 0, mode(me) := satisfied
   ∀ Y ≠ me A(me,Y) := undef
```

4.2 Semantics of $\mathcal{B}_1$

We would like to assume that, in any mode different from *Satisfied*, no customer stalls forever; eventually it makes a move (provided a move is continuously enabled from some time on).

Since in ASMs we have no explicit notion of a move (or program) being enabled, and in partially ordered runs we have no explicit notion of time, both 'enabled' and 'continuously from some time on' need definitions.

There are two obvious candidates for the notion of a program being enabled in a state. One is based on the intuition that a program is enabled if it 'gets down to updates', i.e. if in the given state it generates a nonempty set of updates. The other possibility is that it really changes the state, i.e. that the updateset is nonempty and also nontrivial. We are happy to sidestep the issue here, since for all programs of this paper the two notions will coincide—whenever a nonemtpy set of updates is generated, it will also be nontrivial. Thus we can say that a program is enabled in state σ if it produces a nonempty set of updates in σ.

We say that an agent X stalls forever in a run if (a) X has a last move, say t, and (b) after t a move by X (the program of X) is eventually always enabled (in all $\sigma(J)$ for $J \supseteq I$, for some initial segment $I \ni t$).

We thus assume

Progress Proviso. No customer, in mode other than *Satisfied*, stalls forever.

We consider the runs of $\mathcal{B}_1$ containing enabled moves by customers executing their programs, subject to the Progress Proviso, and also some monitored moves. Our entire knowledge about monitored moves will be encapsulated in explicit requirements D, W1, W2 below.

We now define intervals characterized by the successive executions, by a process X, of its rules Start, Ticket, Entry, Exit, Finale (also in a partial order we refer to open intervals $(a, b) = \{x : a < x < b\}$).

Definition 1. *Suppose a is the move of X executing* Start *rule, and b is the next move by X (which has to execute the* Ticket *rule).*

Then the interval $x = (a, b)$ is a doorway *of X, and $a = \mathsf{Start}(x), b = \mathsf{Ticket}(x)$. If b is the last execution of X then the* wait interval $W(x) = \{t : t > b\}$ *is incomplete and the* CS interval $CS(x)$ *is undefined. Suppose that c is the next move of X after b (necessarily executing* Entry *rule), d is the next move of X after c (necessarily executing* Exit *rule), and e is the next move of X after d (necessarily executing* Finale *rule). Then $W(x) = (b, c)$ and $CS(x) = (c, d)$, $c = \mathsf{Entry}(x)$, $d = \mathsf{Exit}(x)$, $e = \mathsf{Finale}(x)$.*

By Progress Proviso and requirement D below, every doorway is complete, i.e. each execution of Start is followed by execution of Ticket. So is every critical section, i.e. each execution of Entry is followed by executions of Exit (and subsequently Finale).

The program of customer X writes to locations $\mathrm{mode}(X), R(X), A(X,Y)$[2], where locations $A(X,Y)$ with $Y \neq X$ are only cleaned up (that is set to undef) by X (in Finale) and somebody else writes more meaningful information into these locations.

Our program covers all but the reading actions. Since our definitions do not allow 'partially known programs', i.e. a controlled agent can do no more than what his program instructs him to do, more meaningful values have to be written there by the environment, i.e. by somebody else.

We assume that locations $\mathrm{mode}(X), R(X), A(X,X)$ are also controlled by X, i.e. that no other (known or unknown) agent may write there. This is justified by Lamport's algorithm: other customers, as well as the environment, have no business writing to $\mathrm{mode}(X), R(X), A(X,X)$.

This assumption implies that $R(Y)$ is focused, for all customers Y, and, by the program and fact 2, for all moves t in a run of $\mathcal{B}_1$,

Corollary 1. *$R(Y)_{Pre(t)}$ is indisputable iff t compares to all* Start, Ticket *and* Finale *moves of Y.*

To avoid repetitive case distinctions for customers which (being satisfied) have register 0, and of customers which happen to receive the same ticket, we introduce the following notation. If f is a function from customers to natural numbers, let

$$f'(X) = \begin{cases} N \cdot f(X) + \mathrm{id}(X), & \text{if } f(X) > 0; \\ \infty, & \text{otherwise.} \end{cases}$$

Let X, Y range over customers, and x, y over doorways of customers X, Y respectively.

We abbreviate $1 + \max_Y A(X,Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))}$ as $T(x)$.

The declarative requirements, saying what reads need be done, are then (with x being an arbitrary doorway of customer X)

D Each execution of Start by X completes to a doorway x. For each x, for each $Y \neq X$ there is a move $b \in x$, such that $A(X,Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))} = R(Y)_{\mathrm{Pre}(b)}$ (thus $T(x) > R(Y)_{\mathrm{Pre}(b)} \neq \text{undef}$).

W1 If $W(x)$ is complete, then for each $Y \neq X$ there is a move $b \in W(x)$, such that $R(Y)_{\mathrm{Pre}(b)} = A(X,Y)_{\mathrm{Pre}(\mathsf{Entry(x)})}$ (thus $T'(x) < R'(Y)_{\mathrm{Pre}(b)} \neq \text{undef}$).

W2 If $W(x)$ is incomplete, then for some $Y \neq X$ there is an infinite chain $b_1 < b_2 < \cdots$ of moves in $W(x)$, such that, for each n, $R'(Y)_{\mathrm{Pre}(b_n)} < T'(x)$ (thus also $R(Y)_{\mathrm{Pre}(b_n)} \neq \text{undef}$).

D tells us that the value of $R(Y)$, appearing in the array at $\mathsf{Ticket}(x)$, is read in x. W1 says that a permission to go is obtained by executing, for each Y, a successful read in $W(x)$, while W2 tells us that X may be prevented from

[2] by [4] the official notation for these locations is $(\mathrm{mode}, X), (R, X), (A, (X, Y))$; since in the simple cases occurring in this paper, no ambiguity may arise, we shall use the applicative term notation as above also for locations.

going only by executing, for some $Y \neq X$, an infinite sequence of unsuccessful reads in $W(x)$, where a read $b \in W(x)$ from $R(Y)$ on behalf of X is successful if $R'(Y)_{\mathrm{Pre}(b)} > T'(x)$. It turns out that D, W1 and W2 is all that we need to know about reading actions in order to prove correctness and deadlock–freedom of $\mathcal{B}_1$.

Requirements D and W1 say that, for each Y, there is a move $b(Y)$ in x, respectively $W(x)$, having some property. Remark that, without loss of generality, we can assume that these moves $b(Y)$ are all distinct. Suppose namely that, in D or W1, we have $b = b(Y_1) = \cdots = b(Y_k)$. Then we can replace b with k distinct monitored moves which, in the partial order, follow and precede exactly the same moves as b does. It is easy to see that this replacement leaves us with a legitimate run, with exactly the same partial order of customers' moves. A remark of the same kind applies also to sequences of moves claimed by W3, but we shall not need that case.

The reader familiar with [2] might notice that, what in similar requirements there were temporal conditions on some monitored locations, takes here (and in the next section) the shape of conditions on behaviour of unknown agents. The role of some time moments in proofs of [2] thus turns out to be that of place holders for monitored moves.

5 Correctness and Deadlock–Freedom: The ASM $\mathcal{B}_2$

We define an ASM expressing a 'higher level' view of the Bakery Algorithm, similar to $\mathcal{B}_1$ but with the array abstracted away. The relevant datum to be described abstractly is the *ticket* assigned to a customer X (and written into its register $R(X)$) when X leaves the doorway and enters the wait section. We introduce for this purpose two monitored functions, boolean valued Ready and integer valued T, expressing, respectively, readiness of the ticket and its value.

The relevant moment to be analyzed is the moment at which a process with a ticket is allowed to enter the critical section. This 'permission to go' will also be represented by a monitored function, Go.

We will impose requirements on the environment and monitored moves, responsible for the values of Ready, T and Go, which will be shown to guarantee the correctness and deadlock–freedom of the higher level ASM $\mathcal{B}_2$. We will then show that these requirements are correctly implemented in $\mathcal{B}_1$.

5.1 The Program of $\mathcal{B}_2$

Start

```
if mode(me) = satisfied then
   R(me) := 1, mode(me) := doorway
```

Ticket

```
if mode(me) = doorway and Ready(me) then
   R(me) := T(me), mode(me) := wait
```

Entry

```
if mode(me) = wait and Go(me) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   mode(me) := satisfied, R(me) := 0
```

5.2 Semantics of $\mathcal{B}_2$

The ASM $\mathcal{B}_2$ is similar to that of $\mathcal{B}_1$ except for the fact that the array is gone. In particular we assume the Progress Proviso (for known agents, i.e. customers). The role of the array is taken over by three monitored functions, Ready, T and Go. Looking at $\mathcal{B}_1$, Ready(X) and $T(X)$ can be seen as standing for the abbreviations used there, while Go(X) can be interpreted as the guard of the Entry rule, $\forall Y \neq X(A(X,Y) = 0$ or $(A(X,Y), id(Y)) > (A(X,X), id(X)))$.

The ASM $\mathcal{B}_2$ provides however no means to compute Ready, T and Go.

Our first requirement says that every interested customer eventually obtains his ticket:

C0 Each execution of Start, by a customer X, completes to a doorway x. For each x the value $T(X)_{\mathrm{Pre(Ticket(x))}}$ is indisputable.

The indisputable value of $T(X)_{\mathrm{Pre(Ticket(x))}}$ will be, like before, denoted by $T(x)$. In order to express the rest of our conditions on the array in terms of T and Go, we need some additional notation and terminology.

For open intervals in a partial order we also use $(a,b) < (c,d)$ if $b \leq c$, and say that the two intervals are concurrent if neither $b \leq c$ nor $d \leq a$. Note that concurrency does not necessarily imply overlap, i.e. existence of common elements; it in general just allows it.[3]

Sometimes we shall also compare elements with intervals: $c < (a,b)$ if $c \leq a$, likewise for $>$.

This ordering will help us to formalize the idea that tickets increase together with doorways (see C2 below). This should also apply in a way to concurrent

[3] Note however that, if intervals are interpreted as intervals on the partial order of initial segments, with (a,b) containing all segments containing a but not b, then concurrent intervals indeed overlap.

doorways; these are ordered by the following relation $\prec$, borrowed from its linear order analog of [1].

Let $X \neq Y$, and let x, y range over doorways of X, Y respectively.

Definition 2. *$x \lhd y$ if x and y are concurrent and $T'(x) < T'(y)$. Further, $x \prec y$ if $x \lhd y$ or $x < y$.*

Lemma 4. *$x \prec y$ or $y \prec x$.*

Proof. Note that $T'(y) \neq T'(x)$ for $X \neq Y$, while two doorways of the same customer can never be concurrent. □

Our other conditions are then

C1 $T(x)$ is a positive integer > 1.
C2 If $y < x$ then either $\mathsf{Finale}(y) < \mathsf{Ticket}(x)$ or $T'(y) < T'(x)$.
C3 If $W(x)$ is complete, then, for every $Y \neq X$, there exists a move $b \in W(x)$, such that $T'(x) < R'(Y)_{\mathrm{Pre}(b)}$ (thus $R(Y)_{\mathrm{Pre}(b)} \neq$ undef).
C4 If $W(x)$ is incomplete, then there is a $y \prec x$ with $W(y)$ incomplete.

Intuitively, C2 says that tickets respect the temporal precedence of doorways with concurrent wait periods, C4 is an induction principle, and C3 expresses that permission to go is obtained by checking the ticket against competitors' registers. C2 (together with C0) is easily seen to be an abstract version of D, C3 is an abstract version of W1, while the fact, to be proved below, that C4 follows from W2 together with D,W1, is the essence of deadlock–freedom for the Bakery algorithm.

An immediate consequence of C3 is finite concurrency of doorways:

Corollary 2. *The set of doorways concurrent to any given doorway is finite.*

Proof. Let $x < x'$ be two doorways of X both concurrent to y. By C3 applied to x, there is a move b, with $x < b < x'$. Since $R(Y)_{\mathrm{Pre}(b)}$ is indisputable, by corollary 1 b compares to both ends of y; $b \leq \mathsf{Start}(y)$ would imply $x < y$, while $b \geq \mathsf{Ticket}(y)$ would imply $y < x'$, both contradicting the assumption of concurrency. Thus $b \in y$. But, by finite history, there can be only finitely many such b's. □

5.3 Correctness and Fairness of $\mathcal{B}_2$

Lemma 5. (First Comes, First Served) *If $y \prec x$ and $W(x)$ is complete, then $W(y)$ is complete and $CS(y) < CS(x)$.*

Proof. Assume the premise is satisfied and the conclusion is false, i.e. that there is no move $\mathsf{Finale}(y) < \mathsf{Entry}(x)$. Take b as given by C3.

Claim 1 : $T'(y) < T'(x)$.

Claim 2 : $\mathsf{Ticket}(y) < b$.

Given the claims, we have $T'(y) < T'(x) < R'(Y)_{\mathrm{Pre}(b)} \neq$ undef, and thus Y must be writing to $R(Y)$ by a move in $(\mathsf{Ticket}(y), b)$. But the first such write after $\mathsf{Ticket}(y)$ must be a Finale move, which contradicts the assumption that the conclusion of the lemma is false.

Claim 1 follows immediately from definition of $\prec$ in case of concurrency, and from C2 otherwise.

To prove Claim 2, we first note that b is comparable to both ends of y, and, in view of $y \prec x$, $b \leq \mathsf{Start}(y)$ is impossible. It also impossible that $\mathsf{Start}(y) < b \leq \mathsf{Ticket}(y)$, since then $R(Y)_{\mathrm{Pre}(b)} = 1$, which contradicts the choice of b. □

Lemma 6. *$\prec$ is transitive.*

Proof. by contradiction. Suppose $x \prec y \prec z \prec x$. Count the number n of $<$'s in the above sequence of $\prec$ signs. In case $n = 0$ the statement follows from the fact that the order of integers (tickets) is transitive, and in cases $n = 2, 3$ the statement follows the fact that the partial order $<$ of open intervals in a partial order is transitive. In case $n = 1$, by symmetry, we may assume $x \lhd y \lhd z < x$ and therefore $T'(x) < T'(y) < T'(z)$. Given the assumption $x \prec y \prec z \prec x$, by Lemma 5, if one of the waiting sections is complete then so are the other two, and we have $\mathrm{CS}(x) < \mathrm{CS}(y) < \mathrm{CS}(z) < \mathrm{CS}(x)$ which is impossible. So all three waiting sections must be incomplete. Thus we can apply C2 to obtain also $T'(z) < T'(x)$, which is impossible. □

Lemma 7. (Deadlock freedom) *Every $W(x)$ is complete.*

Proof. By corollary 2 (and finite history) $\prec$ is well–founded. Then C4 is precisely the induction principle required to establish the claim. □

This section is summarized in the following

Theorem 1. *Doorways are linearly ordered by $\prec$. All waiting sections are complete, and $x \prec y$ implies $CS(x) < CS(y)$.*

5.4 $\mathcal{B}_1$ Implements $\mathcal{B}_2$ Correctly

We check that the requirements are satisfied in $\mathcal{B}_1$ (i.e. follow from D, W1, W2), where $\mathrm{Ready}(X) = (\forall Y \neq X(A(X, Y) \neq \text{undef}))$, $T(X) = 1 + \max_Y A(X, Y)$, and $\mathrm{Go}(X)$ means that the condition of the rule Entry is satisfied.

C0 is enforced by requirement D and the Progress Proviso for $\mathcal{B}_1$.

C1 is satisfied since the maximum in the rule Ticket is taken over all Y, including X which at that moment has register value $R(X) = 1$.

C2. Take b as given by D. Since $R(Y)_{\mathrm{Pre}(b)}$ is indisputable, the move b compares to all Start, Ticket and Finale moves of Y. With $\mathsf{Ticket}(y) \leq \mathsf{Start}(x) < b$, it is meaningful to ask whether Y executes the Finale move in $(\mathsf{Ticket(y)}, \mathsf{b})$. If it does, we are done; if it doesn't, $R(Y)_{\mathrm{Pre}(b)} = T(y)$, and, by D, $T'(y) < T'(x)$.

C3 follows immediately from W1.

C4. By contradiction, suppose that the premise is satisfied but the conclusion is false, i.e. $W(x)$ is incomplete but $W(y)$ is complete for all $y \prec x$. Let Y and $b_1 < b_2 < \cdots$ be the customer and the sequence of moves as given by W2.

Claim: There is a move $b \in W(x)$, with $R(Y)_{\mathrm{Pre}(b)} \neq$ undef, such that the following two properties hold for each y:

$$\text{(i) } b > \mathsf{Ticket}(y) \qquad \text{(ii) if } y \prec x \text{ then } b > \mathsf{Finale}(y).$$

First we derive the desired contradiction from the claim, and second we prove the claim.

So suppose that the claim is true and let b be as in the claim. Then $R(Y)_{\mathrm{Pre}(b)}$ has an indisputable value, and b thus compares to all moves of Y that change $R(Y)$. What is the value of $R(Y)$ in $\mathrm{Pre}(b)$? We have two possible scenarios. Scenario A: all $y \prec x$; then b succeeds every $\mathsf{Finale}(y)$ and thus $R(Y)_{\mathrm{Pre}(b)} = 0$. Scenario B: there is some y with $\mathsf{Ticket}(y) < b \leq \mathsf{Finale}(y)$; then $R(Y)_{\mathrm{Pre}(b)} = T(y)$. To summarize, if b is as in the claim, then $R(Y)_{\mathrm{Pre}(b)}$ is either 0 or $T(y)$, so that $R'(Y)_{\mathrm{Pre}(b)} \geq T'(y)$.

The values of $R(Y)_{\mathrm{Pre}(b)}$ and of $R(Y)_{\mathrm{Pre}(b_n)}$ for every n are indisputable. Moves b and b_n thus all compare with every move of Y which changes $R(Y)$. It is easy to see that any $b_n \not< b$ satisfies (i) and (ii) in the claim. But then, as shown above, $R'(Y)_{\mathrm{Pre}(b_n)} \geq T'(y)$, which contradicts the property of b_n in W2. Thus every $b_n < b$, which contradicts finite history.

It remains to prove the claim.

To prove the claim, note that, for $y \prec x$, $\mathrm{CS}(y)$ is defined and complete, by the assumption that $W(y)$ is complete and the Progress Proviso. It suffices to prove that there is at most one $y > x$. The sequence of doorways of Y is then finite: by finite history, it has finitely many elements $< x$, by corollary 2 finitely many elements concurrent to x. Thus Y has a last move, say e_Y. By Progress Proviso, e_Y can only be a Ticket or a Finale move. Since all b_n, by corollary 1, compare to e_Y, by finite history, for sufficiently large n we have $b_n > e_Y$. We can then, for claimed b, take any $b_n > e_Y$.

It remains to prove that Y has at most one doorway $> x$. Suppose $x < y$. Then, by C2 (with x, y playing y, x respectively), $T'(x) < T'(y)$ (since $W(x)$ is incomplete). If $W(y)$ were complete, by C3 there would be a $c \in W(y)$ such that $R'(X)_{\mathrm{Pre}(c)} > T'(y)$. But since $x < y < c$, we also have $c \in W(x)$ and $R(X)_{\mathrm{Pre}(c)} = T(x)$, so $T'(x) > T'(y)$, which is impossible. Thus $W(y)$ is incomplete, and y is the last doorway of Y.

We have thus verified that C0–C4 hold of arbitrary runs of $\mathcal{B}_1$. It follows that the results of the previous subsection, summarized in Theorem 1, hold of $\mathcal{B}_1$ as well.

6 Realizing the Model with Reading Agents: the ASM $\mathcal{B}_0$

6.1 The Program of $\mathcal{B}_0$

Until now the universe of known agents was populated by customers only. Now we also have another kind of agents. They present one way of making the unknown agents of $\mathcal{B}_1$ known.

Formally the universe of agents splits into two disjoint universes: Customer and Reader. Customers and readers are related by several functions. If X and Y are distinct customers, then in each state there is at most one reader–agent $r(X, Y)$, and there are no other readers.

If r is the reader $r(X, Y)$, then $\mathrm{Lord}(r) = X$ and $\mathrm{Subject}(r) = Y$. The readers will be created on the fly, when needed (at Start and Ticket moves), and will self–destruct when their task is completed.

Customer Start

```
if mode(me) = satisfied then
   A(me,me) := 1, R(me) := 1, mode(me) := doorway
   ∀ Y ≠ me create-reader(me, Y, doorway)
```

Ticket

```
if mode(me) = doorway and (∀Y ≠me) A(me,Y) ≠ undef then
   A(me,me) := 1 + max_Y A(me,Y), R(me) := 1 + max_Y A(me,Y)
   mode(me) := wait
   ∀ Y ≠ me create-reader(me, Y, wait)
```

Entry

```
if mode(me) = wait and
∀ Y ≠ me (A(me,Y)=0 or (A(me,Y),id(Y)) > (A(me,me),id(me))) then
   mode(me) := CS
```

Exit

```
if mode(me) = CS then
   mode(me) := done
```

Finale

```
if mode(me) = done then
   R(me) := 0, mode(me) := satisfied
   (∀ Y ≠ me) A(me,Y) := undef
```

where create-reader(X, Y, m) abbreviates the rule

```
create r
      agent(r) := true, Reader(r) := true
      program(r) := reader-program
      Lord(r) := X, Subject(r) := Y
      mode(r) := m
endcreate
```

Reader

```
A(Lord(me),Subject(me)) := R(Subject(me))
if mode(me) = doorway then
   destroy-reader(me)
if mode(me) = wait then
   if R(Subject(me)) = 0
      or (R(Subject(me)),id(Subject(me)))
         > (A(Lord(me),Lord(me)),id(Lord(me))) then
   destroy-reader(me)
```

where destroy-reader(a) abbreviates the rule

```
agent(a) := false, Reader(a) := false
program(a) := undef, Lord(a) := undef, Subject(a) := undef
```

6.2 Semantics of $\mathcal{B}_0$

Semantics of $\mathcal{B}_0$ is like that of $\mathcal{B}_1$, but considerably simpler, since all locations are controlled by the known agents, and there are no moves monitored by the known agents, to put constraints on—it is all in the programs for $\mathcal{B}_0$. The reader agents are one way to realize the requirement that those 'for-all commands in the doorway and the wait section of Lamport's pseudocode may be executed in many ways, in various sequences, all at once, concurrently etc.' In fact the reader agents capture all ways to realize that requirement, see below.

The only assumption we have to make, outside of the program, is the Progress Proviso, applying here to all agents, both customers and readers:

Progress Proviso. No reader, and no customer in mode other than *Satisfied*, stalls forever.

The reader–agents are created on the fly, and destroyed upon completion of their task: the effect of destroy-reader(a), if a is a reader–agent, is returning a to the reserve.

6.3 $\mathcal{B}_0$ Realizes $\mathcal{B}_1$

The constraints D, W1, W2 can be read as a rather direct description of what the reader–agents do for their customers in $\mathcal{B}_0$. The fact that every run of $\mathcal{B}_0$ satisfies

D, W1, W2 follows from the programs and the Progress Proviso (together with the semantics described above or in [4]).

D is satisfied in $\mathcal{B}_0$ since, for every move t of X executing Start, for every $Y \neq X$, there is a reader $r(X,Y)$ at Post(Start(x)). By programs and the Progress Proviso each of these readers makes a single self destructive move, which is the b required by D; by programs and the Progress Proviso X eventually executes Ticket.

By programs and Progress Proviso, for every $Y \neq X$ there is a reader $r(X,Y)$ at Post(Entry(x)). That reader makes a move in $W(x)$. For W1, W2 it then suffices to note

Fact 4. *A move t by $r(X,Y)$ in $W(x)$ is the last such move iff it is successful, i.e. $T'(x) < R'(Y)_{Pre(t)}$.*

W1 is namely satisfied in $\mathcal{B}_0$ since, for each $Y \neq X$, we can take the last waiting section move of $r(X,Y)$ for the claimed b.

W2 is satisfied in $\mathcal{B}_0$ since, if all $r(X,Y)$ for $Y \neq X$ have a last move in $W(x)$, by Progress Proviso X must eventually execute Entry. Thus for some $Y \neq X$ the reader $r(X,Y)$ keeps reading forever—take the sequence of his moves for $b_1 < b_2 < \cdots$ as claimed.

We have thus established that every run ρ_0 of $\mathcal{B}_0$ can be viewed as a run of $\mathcal{B}_1$. Since $\mathcal{B}_1$, as far as reading is concerned, can be viewed as a declarative description of algorithmic behaviour, rather then an algorithm proper, ρ_0 can also be seen as a realization of behaviour prescribed by $\mathcal{B}_1$.

To be more precise, let us introduce an appropriate implementation relation between moves and runs of the two ASMs.

A move t_0 by customer X in $\mathcal{B}_0$ *implements* a move t_1 by the same customer in $\mathcal{B}_1$ if the indisputable portions of states (values of $\mathrm{mode}(X), R(X), A(X,X)$) at $\mathrm{Pre}(t_0), \mathrm{Post}(t_0)$ coincide with those at $\mathrm{Pre}(t_1), \mathrm{Post}(t_1)$ respectively.

A run ρ_0 of $\mathcal{B}_0$ *implements* a run ρ_1 of $\mathcal{B}_1$ if the partial order of customers' moves in ρ_0 is order–isomorphic to the partial order of customers' moves in ρ_1, implementing it pointwise: whenever the isomorphism maps a move t_0 in ρ_0 to a move t_1 in ρ_1, then t_0 implements t_1.

In these terms, we have established that $\mathcal{B}_1$ (more specifically, D, W1, W2) provides a sound description of algorithmic behaviour: $\mathcal{B}_0$ is an algorithm behaving like that. For the record,

Lemma 8. (Soundness of $\mathcal{B}_1$) *Each run of $\mathcal{B}_0$ implements a run of $\mathcal{B}_1$.*

We can actually claim more. The requirements D, W1, W2 allow many different behaviours. Is there a behaviour, allowed by $\mathcal{B}_1$, which is not captured by the reader–agents of $\mathcal{B}_0$? Not really. This is the content of the following lemma, expressing a kind of completeness property.

Lemma 9. (Completeness of $\mathcal{B}_1$) *Each run ρ_1 of $\mathcal{B}_1$ is implemented by a run ρ_0 of $\mathcal{B}_0$.*

Proof. The idea is to transform ρ_1 to ρ_0 by implementing reading moves of ρ_1 with appropriate moves of reader–agents, possibly ignoring some inconsequential monitored moves of ρ_1. The replacement process is done in the order of ρ_1, that is earlier read moves are replaced (or discarded) earlier. The following conditions will be guaranteed by induction for every move b introduced by replacement, for every customer X and every doorway x of X. At $\mathrm{Pre}(b)$ there is a reader agent $r = r(X, Y)$ for some $Y \neq X$. If b is a move of r in x, then $\mathrm{mode}(r)_{\mathrm{Pre}(b)} =$ doorway (so that r self–destructs at b), and if b is a move of r in $W(x)$, then $\mathrm{mode}(r)_{\mathrm{Pre}(b)} =$ wait (so that r self–destructs at its last move in $W(x)$).

Now let $x = (a, b)$ be a doorway of X. By D, for each $Y \neq X$ there is a move $b(Y) \in x$ such that $A(X, Y)_{\mathrm{Pre}(\mathsf{Ticket}(x))} = R(Y)_{\mathrm{Pre}(b)}$. Recall that we can assume that $b(Y)$ are all distinct. Then implement each $b(Y)$ with a move of $r = r(X, Y)$. By induction condition, r is in mode doorway before $b(Y)$, and therefore self–destructs there.

The case of read moves in $W(x)$ is similar. Since $W(x)$ is complete, W1 guarantees that, for each $Y \neq X$ there is a move $b(Y) \in W(x)$ such that $R(Y)_{\mathrm{Pre}(b)} = A(X, Y)_{\mathrm{Pre}(\mathsf{Entry}(x))}$. Without loss of generality, all moves $b(Y)$ are distinct. Replace every $b(Y)$ with a move of $r = r(X, Y)$. By the induction condition, r is in mode wait before $b(Y)$, and therefore self–destructs at the move. If $b \in W(x)$ is a monitored move different from all $b(Y)$, we can discard it—also, if there is a $b(Y) > b$, we can implement b with an unsuccessful, nonselfdestructive read of $r(X, Y)$.

Finally, remove all remaining monitored moves in ρ_1. The result is the desired run ρ_0 of $\mathcal{B}_0$, implementing ρ_1. □

7 Concluding Remarks

Corrollary 2 implies the following *cofiniteness* property for the partial runs of the Bakery algorithm: for each move t the set $\{s | s \not> t\}$ is finite. This is a property of the Bakery Algorithm, and not of the modelling framework: in spite of finite history, it is easy to concoct legitimate partially ordered runs of some algorithm which violate the cofiniteness property. In the case of the Bakery Algorithm, the cofiniteness property implies that any two infinitely active customers have to synchronize infinitely many times.

The cofiniteness property is an example of a property of partial runs that is obfuscated in linear runs (since for linears runs it amounts to finite history). This indicates that concurrent computations may have significant properties which cannot be discovered by studying only their linearizations. Concurrent computations should be analyzed directly.

Acknowledgements. We thank Ante Djerek, Robert Eschbach and Igor Urbiha, who have provided very useful remarks on a draft version of this paper.

References

1. Uri Abraham. Bakery algorithms. Unpublished manuscript, pp. 35, 1993.
2. Egon Börger, Yuri Gurevich, and Dean Rosenzweig. The bakery algorithm: Yet another specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 231–243. Oxford University Press, 1995.
3. Paola Glavan and Dean Rosenzweig. Communicating evolving algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, number 702 in Lecture Notes in Computer Science, pages 182–215. Springer, 1993.
4. Yuri Gurevich. Evolving algebra 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
5. Yuri Gurevich and Larry Moss. Algebraic operational semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, number 440 in Lecture Notes in Computer Science, pages 176–192. Springer, 1990.
6. Leslie Lamport. A new solution of Dijkstra concurrent programming problem. *Communications of the ACM*, 17(8):453–455, 1974.

Investigating Java Concurrency Using Abstract State Machines

Yuri Gurevich[1], Wolfram Schulte[1], and Charles Wallace[2]

[1] Microsoft Research, Redmond, WA 98052-6399, USA
{gurevich,schulte}@microsoft.com
[2] University of Delaware, Newark, DE 19716, USA
wallace@cis.udel.edu

Abstract. We present a mathematically precise, platform-independent model of Java concurrency using the Abstract State Machine method. We cover all aspects of Java threads and synchronization, gradually adding details to the model in a series of steps. We motivate and explain each concurrency feature, and point out subtleties, inconsistencies and ambiguities in the official, informal Java specification.

1 Introduction

The *Java* programming language [7,13] provides sophisticated support for concurrency. The fundamental operations of concurrent programs are implemented as built-in features of the language. Many of the notoriously complicated details of concurrent programming are hidden from the programmer, simplifying the design of concurrent applications. Furthermore, a platform-neutral memory model is included as part of the language. The incorporation of such intricate, subtle operations into Java calls for a precise specification. As interest in the language's concurrency model increases, Java developers are examining and exploiting its details [15,17,16,14]. The popularity of Java and, more importantly, its emphasis on cross-platform compatibility make the need for such a specification even stronger.

We present a model of the concurrent features of Java, using the formal operational specification method of *Abstract State Machines (ASMs)*[1] [9,22,23]. We use the *Java Language Specification* manual *(JLS)* [7], as our reference for the language. The JLS is an informal specification, and due to the ambiguity which pervades natural language, it can be interpreted in different ways. Our model gives an unambiguous specification which reflects our interpretation of the JLS. Throughout the paper, we indicate where ambiguities and omissions in the JLS give rise to other interpretations. The formal specification process also uncovers many subtle but important issues which the JLS does not bring to light. Our goal is a specification that is not only precise but accessible to its readers, even those not familiar with Java, concurrent programming, or ASMs. As part of this

[1] ASMs were formerly known as *Evolving Algebras*.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 151–176, 2000.

project, we implement the ASM specifications using the *ASMGofer* interpreter [20]. This implementation serves as a convenient tool for prototyping and testing.

It is important to distinguish unintentional ambiguity from intentional underspecification. The authors of the JLS envision support for Java concurrency taking many different forms: "by having many hardware processors, by time-slicing a single hardware processor, or by time-slicing many hardware processors [7]". The JLS leaves some details unspecified in order to give implementers of Java a certain amount of freedom: "the intent is to permit certain standard hardware and software techniques that can greatly improve the speed and efficiency of concurrent code [7]". As is usual with the ASM methodology [8,2], we design our ASMs to model Java concurrency at its natural level of abstraction, capturing the essence of the JLS without committing to any implementation decisions.

While most of the JLS specification is imperative in nature, some of the concurrency model is described in a declarative style: rather than explain how to execute a concurrent program, the JLS gives conditions that a correct execution must meet. This shift in presentation style is most likely due to the notion that only a declarative specification can be truly implementation-independent.

There are two ways for us to deal with the declarative portions of the JLS. The obvious and simple way is to reformulate them declaratively as conditions on the execution of our ASM. The other way is to implement them in the ASM itself. We choose to do the latter, whenever it is natural to do so, but in a way that does not sacrifice generality. Our model obeys the conditions established in the JLS, yet it is fully general in the sense that for any concurrent program execution that follows the rules of the JLS, there is a corresponding run of the ASM. We find our imperative approach helpful in creating a clear mental picture of the concurrency model, and in uncovering hidden assumptions, inconsistencies and ambiguities in the JLS.

There are several formalizations of Java concurrency in the literature. Börger and Schulte [3] also uses ASMs, to give a semantic analysis of Java that exposes a hierarchy of natural sublanguages. It covers the language features for concurrency, but its focus is on the high-level language rather than the lower-level concurrency model, so it does not provide a full specification of Java concurrency. There are several works [1,4,5] using the *Structural Operational Semantics (SOS)* methodology [18]. Attali *et al.* [1] do not concentrate on the lower-level details of Java concurrency, while Cenciarelli *et al.* [4] give them in a declarative style, in keeping with the presentation in the JLS. As mentioned earlier, we believe that there are advantages to our imperative approach. Coscia and Reggio [5] explicitly deal with the details of the concurrency model and propose some changes to its design. Gontmakher and Schuster [6] present a non-imperative specification, with the purpose of comparing Java's memory behavior with other well-known notions of consistency. Our focus is different, however: while their work assumes a particular interpretation of the JLS and proceeds from there, our goal is first to come to a clear understanding of the concurrency model as presented in the

JLS, and then to discuss its consequences. As we shall see, there are several issues within the JLS which make this initial understanding difficult.

We introduce in §2 the basic rules by which agents (*threads*) interact with regard to shared *variables*, and in §3 the special considerations for variables marked as *volatile*. In §4 we introduce *locks*, which are used to limit concurrency within a program. In §5 we discuss *prescient* store actions. In §6 we describe *thread objects*. In §7, we cover *waiting* and *notification*.

Our technical report [11] provides a complete presentation of our ASM specification. In it, we also prove that our specification is as general as possible while obeying all the correctness criteria of the JLS.

Acknowledgment. This work was partially supported by NSF grant CCR-95-04375.

2 Threads and Variables

In a *single-threaded* computation, a single agent executes the instructions of a program, one at a time. In contrast, a run of a Java program may be *multithreaded*, (intuitively) involving multiple agents that execute instructions concurrently. Each agent executes sequentially, the sequence of its actions forming a *thread* of execution. In the parlance of distributed computing, the term "thread" is used to refer not only to an agent's computation but also to the agent itself. Threads may execute on a single processor or multiple processors. The atomic actions of different threads may be interleaved; they may even be concurrent in the case of multiple processors.

Since different threads may access and update common data, the order of their actions affects the results of an execution. The JLS establishes some conditions on the interaction of threads with respect to shared data, but intentionally leaves some freedom to implementers of the language. Consequently, the results of executing a multithreaded program may vary between different Java platforms, and even between different executions on the same platform.

A data value is either an instance of a primitive type, such as `int` or `boolean`, or an *object*, a dynamically created value. A value resides at a location either in the *working memory* of a thread or in the *main memory*. Certain memory locations serve as manifestations of a *variable*. The *master copy* of a variable is the unique location in the main memory that manifests the variable. A thread's *working copy* of a variable is the unique location in the thread's working memory that manifests the variable. A thread accesses or updates only its working copies.

The JLS speaks of the main memory as an agent which communicates with threads and updates variable master values. While this view is adequate for describing a system with a centralized main memory, it does not capture the possibility of a distributed main memory [16], where updates of different variable master copies may occur concurrently. We find it convenient to imagine that each variable has a *master* agent, which accesses and updates the variable's master copy. Since there is a one-to-one correspondence between variables and their masters, we may identify a variable master with the variable it controls.

We create an ASM to model threads' actions on variables. The agents of are threads (members of Thread) and variable masters (members of Var). At the beginning of each run, there is a single member of the universe Thread. Since we identify variable masters with the variables they control, a member of the universe Var is both a variable and a variable master. The universe Value represents the set of values that master or working copies of variables may bear. The function masterValue maps each variable to the current value of its master copy. The function workingValue, given a thread and variable, returns the current value of the thread's working copy of that variable.

A thread's *execution engine* is the component of the thread that executes the Java program. It may perform actions on variables: *creating* a variable, *assigning* a new value to a variable, or *using* a previously assigned value of a variable. There may be more than one value associated with a variable, but a thread has immediate access only to the value of its working copy. A use or assign action is internal to a thread, involving only its working copy. Since we are not interested here in what a thread computes during its execution, our view of use and assign actions is simple: an assign action just changes the value of a variable's working copy, and a use action does nothing. (Although what happens in a use action is not important to us, we shall see in §3 that the *occurrence* of a use action may be significant, even at our level of abstraction.) A thread's execution engine may do one other sort of action of interest to us: creating another thread, which may execute concurrently with its creator.

Threads pass values of shared variables among themselves via the main memory. A thread may update the master copy of a variable with a freshly assigned value, or it may request the current value of the master copy. This is done through asynchronous communication with a variable's master agent. To transfer the value of its master copy to a thread's working memory, (the master of) a variable issues a *read* action. This is followed by a *load* action by the thread, which installs the value into its working memory. To transfer its working value of a variable to the main memory, a thread issues a *store* action. This is followed by a *write* action by (the master of) the variable, which installs the value into the main memory.

Somehow information is passed from variable to thread in a read-load sequence, and from thread to variable in a store-write sequence. To give a more imperative character to our description, without any essential loss of generality, we introduce an explicit but quite abstract notion of *messages* passing between threads and variables. When a variable issues a read action, it sends an *access message* to a thread, and the target thread then receives the message by issuing a load action. When a thread performs a store action, it sends an *update message* to a variable, and the variable then receives the message by issuing a write action. So for example, we may speak of a thread storing out to a variable by sending it an access message, or a variable writing in from a thread by installing the value of an update message.

We define a universe Msg comprising the universe AccMsg (messages from variables to threads) and the universe UpdMsg (messages from threads to varia-

bles). The function var maps each access message to the variable that sent it, and maps each update message to the variable that is its intended recipient. The function thread maps each update message to the thread that sent it, and maps each access message to the thread that is its intended recipient. The function value returns the value contained in a given message.

In any state, the members of the universe AccMsg that have been sent from a variable v to a thread t form a "subuniverse" of AccMsg, which we call AccMsg(v, t). Likewise, the members of the universe UpdMsg that have been sent from t to v form a subuniverse of UpdMsg, which we call UpdMsg(t, v).

While the JLS does not dictate a specific policy on the access of shared variables, it does impose some rules on the concurrent behavior of threads and variables. We present these rules as they appear in the JLS and give our interpretations of them.

JLS rules 1

1. [p. 403] "The actions performed by any one thread are totally ordered; that is, for any two actions performed by a thread, one action precedes the other."
2. [p. 403] "The actions performed by the main memory for any one variable are totally ordered; that is, for any two actions performed by the main memory on the same variable, one action precedes the other."
3. [p. 403] "It is not permitted for an action to follow itself."

In other words, actions on variables form a partial order (which we denote by $<$) in which the actions of any single thread are linearly ordered, and the actions of any single variable master are linearly ordered. Note that this supports our view of independent variable master agents as opposed to a monolithic main memory. The read and write actions on a single variable are ordered linearly, but read and write actions on different variables may occur concurrently. This follows naturally if we think of the master copy of each variable as controlled by an independent agent.

JLS rules 2

1. [p. 403] "Each load action by a thread is uniquely paired with a read action by the main memory such that the load action follows the read action."
2. [p. 403] "Each store action by a thread is uniquely paired with a write action by the main memory such that the write action follows the store action."
3. [p. 405] "For every load action performed by any thread t on its working copy of a variable v, there must be a corresponding preceding read action by the main memory on the master copy of v, and the load action must put into the working copy the data transmitted by the corresponding read action."
4. [p. 405] "For every store action performed by any thread t on its working copy of a variable v, there must be a corresponding following write action by the main memory on the master copy of v, and the write action must put into the master copy the data transmitted by the corresponding store action."

5. [p. 405] "Let action A be a load or store by thread t on variable v, and let action P be the corresponding read or write by the main memory on variable v. Similarly, let action B be some other load or store by thread t on that same variable v, and let action Q be the corresponding read or write by the main memory on variable v. If A precedes B, then P must precede Q."

These rules restrict the ways in which a thread's load and store actions may be interleaved with a variable's read and write actions. First, we note that the term "uniquely paired" in Rules 2.1 and 2.2 is ambiguous. A statement of the form "Each element x of X is uniquely paired with an element y of Y such that $\phi(x, y)$" has a lenient interpretation: "For all x in X there is a unique y in Y such that $\phi(x, y)$". It also has a strict interpretation: "For all x in X there is a unique y in Y such that $\phi(x, y)$, and for all y in Y there is a unique x in X such that $\phi(x, y)$". The lenient interpretation allows *spurious* y's; that is, y's that are not paired with any x. Which interpretation shall we use? The strict interpretation is what the authors of the JLS intended [21], so we adopt it here. However, we find that a lenient interpretation of the term in Rule 2.1 has some advantages that make it worth discussing.

Read/load order. For every load action L by a thread t on a variable v, there is a read action $R < L$ by v such that L loads in the value read out at R (Rules 2.1 and 2.3). This supports our conception of access messages: every load action must load in the value of an access message issued by a previous read action. Furthermore, a thread may not load in the same access message twice (Rule 2.5).

The choice of interpretation of "uniquely paired" determines whether every access message must be loaded in. The strict interpretation, under which every access message is loaded in, is simpler and more appealing from a logical perspective, which is why it is the intended interpretation [21]. But note that spurious read actions are innocuous: failing to load in a given access message has only the effect of making a thread's working memory less up-to-date than possible. Furthermore, the lenient interpretation allows a higher degree of independence of threads and variables: a variable is free to issue access messages without concern for whether they are eventually loaded in. For instance, updated master values of an important variable could be broadcast to threads, with each thread deciding whether to load the value in; or blocks (*pages*) of master values could be sent to a thread, with the thread selecting which values to load in.

Store/write order. For every store action S by a thread t on a variable v, there is a write action $W > S$ by v such that W writes in the value stored out at S (Rules 2.2 and 2.4). Using our message-passing parlance, every store action sends an update message which must be written in by a following write action.

As with the case of read and load actions, how we interpret "uniquely paired" determines whether every write action is an action of writing in an update message. Here, adopting the strict interpretation seems less controversial. It is not clear what value a spurious write action writes. At best, it simply rewrites the value that is already in main memory, in which case it is useless. But if it writes

a different value, it may obliterate a current value in main memory, replacing it with an outdated or invalid value.

Read/write order follows load/store order. Each thread performs load and store actions on a variable in a certain order, and the variable must perform the corresponding read and write actions in the same order. Let us examine this more closely.

Load/load. If a thread t performs a load action L_1 (with corresponding read action R_1) on a variable v, and another load action L_2 (with corresponding read action R_2) on v, Rule 2.5 postulates that $(L_1 < L_2) \Rightarrow (R_1 < R_2)$. If v sends access messages m and m' to t, then t may load in m before m' only if m was sent before m'. In other words, t loads in ever newer access messages from v. Without loss of generality, we may think of t as discarding m when it loads it in.

The conjunction of Rule 2.3 and Rule 2.5 implies Rule 2.1. Rule 2.1 asserts that each load action has a preceding unique read action. Rule 2.3 asserts that every load action has a preceding read action and Rule 2.5 asserts that different load actions must have different corresponding read actions, ensuring uniqueness.

Store/store. If a thread t performs a store action S_1 on v (with corresponding write action W_1) and another store action S_2 on v (with corresponding write action W_2), Rule 2.5 postulates that $(S_1 < S_2) \Rightarrow (W_1 < W_2)$. If t sends an update message m to v before sending another update message m' to v, then m must be written before m'. In other words, v writes in ever newer values stored out by t. Thus writing m' is allowed only if m has been delivered through a write action. We can think of v as discarding m when it writes it.

Store/load. If a thread t performs a store action S (with corresponding write action W) on v, and t performs a load action L (with corresponding read action R) on v, Rule 2.5 postulates that $(S < L) \Rightarrow (W < R)$. In other words, if t sends an update message m to v before loading an access message m' in from v, then v must write m in before sending m'. If t stores its working value out to v and then immediately loads in from v, and other threads do not store out to v in the meantime, then the value it receives from the load is the value it stored.

Thus sending m' is allowed only *after* the point in time when m is written in. If m' is sent but not loaded in *before* this point in time, it can never be loaded in, and thus Rule 2.1 is violated (under the strict interpretation). To avoid this, v must not issue an access message to t as long as there is a pending update message from t to v, and t must not issue an update message to v as long as there is a pending access message from v to t. Furthermore, v and t must be prevented from sending messages to each other concurrently.

Under the assumption that messages take no time to travel, we may require that v checks for incoming update messages from t before sending an access message to t, and t checks for incoming access messages from v before sending an update message to v. But this does not eliminate the possibility of concurrent actions by v and t. Thus some means of avoiding concurrent actions by v and t is necessary. Note that this complication disappears if we were to adopt the lenient

interpretation of "uniquely paired" for the pairing of read and load actions. In our example, t could simply ignore the access message m' and still have its update message m written.

To represent the relative age of messages, we introduce the relation $<$, and the term oldest? to determine whether a given message has no predecessors.

term oldest?(m): (not $\exists m'$: Msg) $m' < m$

The JLS does not impose any particular means of ordering messages, so we avoid doing so here by making the function external. We restrict attention to runs in which the function $<$ behaves in the expected way.

JLS rules 3 Let t be a thread and v be a variable.

1. [p. 404] "A use or assign action by t of v is permitted only when dictated by execution by t of the Java program according to the standard Java execution model."
2. [p. 404] "A store action by t on v must intervene between an assign by t of v and a subsequent load by t of v."
3. [p. 404] "An assign action by t on v must intervene between a load or store by t of v and a subsequent store by t of v."
4. [p. 404] "After a thread is created, it must perform an assign or load action on a variable before performing a use or store action on that variable."
5. [p. 405] "After a variable is created, every thread must perform an assign or load action on that variable before performing a use or store action on that variable."

These rules impose constraints on the exchange between a thread's execution engine and working memory.

Use and assign actions. Threads may only issue use and assign actions when needed for progress by the execution engine (Rule 3.1). At the level of abstraction we have chosen, we are not concerned with what threads actually compute, so this rule does not affect our model. Nevertheless, we should note that the term "dictated" is somewhat ambiguous here. For example, let `var` be an expression that requires a use action on a variable v. In evaluating the expression `(var + var)`, how many use actions on v are dictated by the program? The strictest interpretation would require a use action for each reference to a variable (our example would require two use actions), and this is what the authors of the JLS intended [21].

Store actions. If a thread t performs a load or store action LS on v and then performs a store action S on v, Rule 3.3 postulates that t performs an assign action A on v such that $LS < A < S$. In other words, spurious store actions are forbidden: t may only store out to v if it has a new value to transmit. This prevents the current contents of the main memory from being overwritten by older (*i.e.*, previously stored) values. Without this rule, a thread could store out outdated values it loaded in long ago, overwriting more current values that other threads stored out in the interim.

If a thread t performs an assign action A on a variable v and then performs a load action L on v, there is a store action S by t on v such that $A < S < L$ (Rule 3.2). In other words, a thread must store out a value of a variable v after issuing a sequence of assigns to v, transmitting at least the last assigned value to the main memory, before attempting a load; the assigns are not completely forgotten. A thread may load an access message in from a given variable only if the last working value it assigned has been stored out. Note that a thread may assign to a variable and then use the variable with no intervening load. In this case, no intervening store is required, and the assigned value may simply remain in the working memory.

This makes the coordination between threads and variables more complex. Once t performs an assign on v, its next action on v cannot be a load. So if t assigns a value to v concurrently with v sending an access message to t, t may not subsequently load the message in until it stores its working value out to v. But it cannot perform this store action, as the write action corresponding to the store would follow v's read action; thus Rules 2.1 and 2.3 are violated. To avoid this, v must not issue an access message to t if there is an assign action by t on v that has not been followed by a store action on v, and t must not assign to v if there is a pending access message. So the actions which threads and variables must avoid doing concurrently are not reads and stores, as Rule 2.5 might suggest, but reads and assigns.

Use/store actions preceded by an assign/load action. If a thread t performs a use or store action US on a variable v, Rules 3.4 and 3.5 postulate that there is an assign or load action $AL < US$ by t on v. For use and store actions to behave properly, there must be a valid value in working memory. Initially, all the contents of a thread's working memory are invalid, but as a result of these rules, no invalid value of the working memory is ever used or stored out.

We define the following functions to help enforce the conditions described above. Rule 3.3 allows a thread t to store out its working value of a variable v only if it has assigned a fresh value to v but has not stored it out; Rule 3.2 allows t to load an access message in from v only if it has no freshly assigned working value of v to store out. The function freshAssign? determines whether a given thread has assigned a fresh value to a given variable without storing it out. Rules 3.4–5 allow t to use its working value of v only if there is a valid value of v in its working memory, installed there by a load or assign action. The function usableValue? determines whether a given thread has a valid value for a given variable in its working memory.

The actions of a variable master consist of issuing a read message to a thread, and writing in an update message.

module Var: **choose among**
  choose t: Thread: readOK?(t)
    Read out to t
  choose t: Thread
    choose m: UpdMsg(t, Self): writeOK?(m, t)
      Write m in from t

rule *Read out to t*: **extend** AccMsg(Self, t) **with** m
value(m) := masterValue(Self)

rule *Write m in from t*: masterValue(Self) := value(m)
UpdMsg(m) := false

The terms readOK? and writeOK? determine whether the given action is allowed by the JLS rules. Rules 2.5 and 3.2 allow a variable v to read out to a thread t only if every assign action by t on v has been followed by corresponding store and write actions. Rule 2.5 allows t to write a message in from v only if the message is the oldest pending message from t to v.

term readOK?(t): not (freshAssign?(t, Self) or ($\exists m$: UpdMsg(t, Self)))

term writeOK?(m, t): oldest?(m)

The actions of a thread consist of storing a value out through an update message, loading in an access message, or taking a step in executing the Java program. Program execution may involve using the working value of a variable, assigning a value to the working copy of a variable, creating a variable, or creating a thread.

module Thread: **choose among**
Execute program *WM-MM transfer*

rule *Execute program*: **choose among**
WM-EE transfer *Create var* *Create thread*

rule *WM-EE transfer*: **choose among**
choose v: Var: useOK?(v)
Use v
choose v: Var: assignOK?(v)
Assign to v

rule *WM-MM transfer*: **choose among**
choose v: Var
choose m: AccMsg(v, Self): loadOK?(m, v)
Load m in from v
choose v: Var: storeOK?(v)
Store out to v

We define rules for load, store, use and assign actions, as well as the actions of creating a variable and creating a thread.

rule *Load m in from v*: workingValue(Self, v) := value(m)
usableValue?(Self, v) := true
AccMsg(m) := false

rule *Store out to v*: freshAssign?(Self, v) := false
extend UpdMsg(Self, v) **with** m
value(m) := workingValue(Self, v)

rule *Use* v: **skip**

rule *Assign to* v: freshAssign?(Self, v) := true
usableValue?(Self, v) := true
choose val: Value
workingValue(Self, v) := val

rule *Create var*: **create** Var **agent** v

rule *Create thread*: **create** Thread **agent** t

The terms loadOK?, storeOK?, useOK? and assignOK? determine whether the given action is allowed by the JLS rules. Rule 2.5 allows a thread t to load a message in from a variable v only if the message is the oldest pending message from v to t. Rules 3.3–5 allow t to store out to v only if there has been an assign action by t on v without a following store action. Rules 3.4–5 allow t to use v only if there has been an assign or load action that put a value in t's working copy of v. Rules 2.5 and 3.2 allow t to assign to v only if every access message from v to t has been loaded in.

term loadOK?(m, v): oldest?(m) and not freshAssign?(Self, v)

term storeOK?(v): freshAssign?(Self, v)

term useOK?(v): usableValue?(Self, v)

term assignOK?(v): (not $\exists m$: AccMsg(v, Self))

3 Volatile Variables

The basic model for threads and shared variables, as presented in §2, permits optimizations that reduce the amount of communication between agents and thus enhance the performance of multithreaded programs.

Communication between threads and the main memory can be lessened through *caching*: keeping values in working memory without transmitting them to main memory or getting new values from main memory. Instead of consulting the main memory every time it uses a variable, a thread may service several use actions on the same variable with a single load action. It may also service several assign actions with a single store action, sending only the last of these assigned values to main memory.

Caching may have undesirable results, particularly for variables that are assigned to and used frequently by different threads. Cached values may become outdated as other threads store out to the main memory. Also, caching assigned values prevents other threads from viewing the newly assigned values as they arise.

Communication between variables can be avoided altogether, allowing them to operate independently of one another. While Rule 2.5 dictates that the order of a thread's actions on a *single* variable is also followed by the variable, the order

of its actions over *different* variables need not be followed in the main memory. Consequently, variables need not coordinate their actions among themselves.

Hence for certain variables, a stricter discipline is necessary. Java allows the programmer to declare variables *volatile* at the time of their creation. Volatile variables follow a policy that disallows the optimizations described above.

We modify the ASM to model operations on volatile variables. We add a universe VolVar, representing the set of volatile variables. Every member of VolVar is also a member of Var. The rule *Create var* requires a slight change: a new variable may be marked as volatile.

The JLS imposes the following additional conditions on volatile variables.

JLS rules 4 [p. 407] "Let t be a thread and let v and w be volatile variables."

1. "A use action by t on v is permitted only if the previous action by t on v was load, and a load action by t on v is permitted only if the next action by t on v is use. The use action is said to be "associated" with the read action that corresponds to the load."
2. "A store action by t on v is permitted only if the previous action by t on v was assign, and an assign action by t on v is permitted only if the next action by t on v is store. The assign action is said to be "associated" with the write action that corresponds to the store."
3. "Let action A be a use or assign by thread t on variable v, let action F be the load or store associated with A, and let action P be the read or write of v that corresponds to F. Similarly, let action B be a use or assign by thread t on variable w, let action G be the load or store associated with B, and let action Q be the read or write of v that corresponds to G. If A precedes B, then P must precede Q."

Caching is disallowed. If a thread t performs a load action L on v, Rule 4.1 postulates that there is a use action $U > L$ by t on v, and there is no action *Act* by t on v such that $L < Act < U$. In other words, each value loaded in from the main memory is used exactly once. In contrast, multiple use actions on a non-volatile variable may use a value loaded in by a single load action.

Rule 4.2 postulates that a thread t performs a store action S on a volatile variable v if and only if there is an assign action $A < S$ by t on v and there is no action *Act* by t on v such that $A < Act < S$. In other words, every assigned value must be stored out and written to the main memory exactly once. For non-volatile variables, assign actions may overwrite one another without having their values stored out.

Read/write order follows use/assign order. Each thread performs use and assign actions on volatile variables in a certain order; Rule 4.3 ensures that the variables must perform the corresponding read and write actions in the same order. This condition is similar to Rule 2.5, but different in some important ways. First, the thread actions mentioned in Rule 4.3 are use and assign actions, as opposed to store and load actions as in Rule 2.5. Also, the ordering holds over actions on all volatile variables, not just those on a single variable.

To account for the behavior of volatile variables, we modify our view of message passing between threads and variables. We think of a volatile variable access message as ending in a use action, as opposed to a load action. Similarly, we think of a volatile variable update message as originating with an assign action, as opposed to a store action. By Rules 2.4 and 4.2, all volatile update messages must be stored out and written, and by Rules 2.3 and 4.1, all volatile access messages must be loaded in and used. We examine this more closely.

Use/Use. If a thread t performs a use action U_1 (with corresponding read action R_1) on a volatile variable v, and another use action U_2 (with corresponding read action R_2) on a volatile variable w, Rule 4.3 postulates that $(U_1 < U_2) \Rightarrow (R_1 < R_2)$. Note that R_1 and R_2 may be actions by different variables. Given an access message m from v and an access message m' from w, t may use m before m' only if m was sent before m'. In other words, t uses ever newer volatile access messages.

The conjunction of Rule 4.3 with Rules 4.1 and 3.1 implies that volatile variables must work closely with threads' execution engines. Once a thread loads in a read message from a volatile variable, by Rule 4.1 it must use that variable, and by Rule 4.3 it must do so before performing any other assign or use action. Rule 3.1 forbids use actions not "dictated" by the execution engine, so the message must be sent with knowledge of what is needed by the Java program. Consider a scenario where a thread has just loaded an access message from a volatile variable v, but the next operation by the execution engine is an assign action on v, or an action on another volatile variable. The only way for t to progress is to issue a gratuitous use action on v, but this is forbidden by Rule 3.1. Therefore, volatile variables must be careful when issuing access messages to a thread, doing so only when it is clear that the thread's execution needs one immediately.

Rule 4.3 also implies that volatile variables must not read out to the same thread concurrently. As the actions of a single thread are linearly ordered, by Rule 4.3 the corresponding read actions must be linearly ordered as well. This may require some coordination between volatile variables; if several volatile variables wish to read out to a single thread, only a single variable at a time may do so.

Assign/Assign. If a thread t performs an assign action A_1 (with corresponding write action W_1) on a volatile variable v, and another assign action A_2 (with corresponding write action W_2) on a volatile variable w, Rule 4.3 postulates that $(A_1 < A_2) \Rightarrow (W_1 < W_2)$. Note that W_1 and W_2 may be actions by different variables. In other words, given an update message m from t to v and an update message m' from t to w, v may write m before m' only if m was sent before m'. So volatile update messages sent by t are written in the order in which they are sent.

Assign/Use. If a thread t performs a use action U (with corresponding read action R) on a volatile variable v, and an assign action A (with corresponding write action W) on a volatile variable w, Rule 4.3 postulates that $(A < U) \Rightarrow$

($W < R$). In other words, if t sends a volatile update message m before using a volatile access message m', then m must be written in before m' is sent.

Thus sending m' is allowed only *after* the point in time when m and all other volatile update messages from t have been written. If m' is sent but not used *before* this point in time, it can never be used, hence violating Rule 4.1. To avoid this, v must not issue an access message to t as long as there is a pending volatile update message from t, and t must not issue a volatile update message as long as there is a pending volatile access message to t. Furthermore, volatile access and update messages must not be sent to and from t concurrently.

We define some additional functions to help enforce these conditions. Rule 4.1 requires a thread t to use a volatile variable v if and only if it has loaded in a message from v without using its value. If a thread has loaded in a message from a volatile variable but not used its value, the function msgToUse returns this message. If a thread has assigned a value to a volatile variable but not stored the value out, the function msgToStore returns the volatile update message.

The terms readOK? and writeOK? include extra conditions on volatile variables. Rule 4.3 allows a volatile variable v to read out to a thread t only if all assign actions by t on volatile variables have been followed by corresponding store and write actions. To readOK? we add the conjunct VolVar(Self) $\Rightarrow$ ($\forall v$: VolVar) (not $\exists m$: UpdMsg(t, v)). Rule 4.3 allows v to write an update message from t in only if the message has been stored out and is the oldest pending volatile message from t. To writeOK? we add the conjunct VolVar(Self) $\Rightarrow$ $m \neq$ msgToStore(t, Self).

We modify the rules for load and store actions. A volatile access message is removed when it is used, rather than when it is loaded in. A volatile update message is created through an assign action, rather than through a store action.

rule *Load m in from v*: workingValue(Self, v) := value(m)
usableValue?(Self, v) := true
if VolVar(v) **then** msgToUse(Self, v) := m
else AccMsg(m) := false

rule *Store out to v*: freshAssign?(Self, v) := false
if VolVar(v) **then** msgToStore(Self, v) := undef
else extend UpdMsg(Self, v) **with** m
value(m) := workingValue(Self, v)

The term loadOK? includes an extra condition on volatile variables. Rule 4.1 allows a thread t to load a message in from a volatile variable v only if it has used all previous loaded access messages from v. Rule 4.2 allows t to load a message in from v only if it has stored out all previously assigned values to v. To loadOK? we add the conjunct VolVar(v) $\Rightarrow$ not usableValue?(Self, v).

We modify the rules for use and assign actions. A use action on a volatile variable removes the volatile access message from the AccMsg universe. An assign action on a volatile variable generates a UpdMsg.

```
rule Use v: if VolVar(v) then
                usableValue?(Self, v) := false
                msgToUse(Self, v) := undef
                AccMsg(msgToUse(Self, v)) := false

rule Assign to v: freshAssign?(Self, v) := true
                  if not VolVar(v) then usableValue?(Self, v) := true
                  choose val: Value
                      workingValue(Self, v) := val
                      if VolVar(v) then
                          extend UpdMsg(Self, v) with m
                              val(m) := val
                              msgToStore(Self, v) := m
```

The terms useOK? and assignOK? include extra conditions on volatile variables. Rule 4.1 allows a thread t to use its working value of a variable v only if it has loaded in a message from v without using its value. Rule 4.3 requires this message to be the oldest pending volatile access message to t. To useOK? we add the conjunct VolVar(v) $\Rightarrow$ oldest?(msgToUse(Self, v)). Rule 4.2 allows t to assign a value to v only if all of t's previous assigns to t have been stored out, and Rule 4.3 requires that there be no pending volatile access messages to t. To assignOK? we add the conjunct VolVar(v) $\Rightarrow$ not (freshAssign?(Self, v) or usableValue?(Self, v) or ($\exists m$: AccMsg(v', Self))).

If two volatile variable variables read out to t concurrently, the corresponding load actions will be ordered, since t can only perform one load action at a time. But the read actions will not be ordered, thereby violating Rule 4.3. We restrict attention to runs in which this does not occur.

Rules 4.1 and 4.2 ensure that if a load or assign action on a VolVar occurs, a corresponding use or store action will occur sometime in the future. Similarly to Rules 2.1 and 2.2, these rules can be flouted by delaying a use or store action indefinitely. We restrict attention to runs that avoid this situation.

4 Locks

Certain situations require that a thread be able to perform a series of operations without interference from other threads. In general, the programs of threads $t_1 \ldots t_n$ may have critical regions that they should avoid executing concurrently. It seems reasonable to try to prevent any t_i and t_j from entering their critical regions simultaneously. A natural way to solve this problem in an object-oriented framework is to have one critical *object* related to all these critical regions. All the threads may refer to the critical object, but only one thread may own the object at any given time, and only the owner can execute its critical region. Java provides support for this sort of mutual exclusion. The critical regions are called *synchronized* regions.

In Java, each object has a unique *lock* associated with it. A thread gains entry into a synchronized code region of its program by performing a *lock* action

on the lock of the critical object. The thread then *holds* the lock until it exits the synchronized code region and performs an *unlock* action on the lock.

Various threads may issue lock or unlock actions on the same lock. The lock and unlock actions on a single lock are performed in conjunction with an arbiter, which restricts the number of threads holding a single lock to one at a time. The JLS speaks of main memory as this arbiter. As with variable master copies, we prefer to think of each lock as controlled by a *master* agent, rather than by the (possibly distributed) main memory.

We modify the ASM to model operations on locks. The agents are threads, variable masters, and lock masters. We represent locks as members of the universe Lock. As with variables, we identify lock master agents with the locks they control, so a member of the universe Lock is both a lock and a lock master. Objects are members of the universe Object. A LockActionType is either lock or unlock, and a LockAction is a pair consisting of a LockActionType and a Lock on which to perform the given action. The function lock maps each object to its lock.

To see how the introduction of locks ensures that only one thread enters its synchronized region, we must consider the JLS rules for the concurrent behavior of locks.

JLS rules 5

1. [p. 403] "The actions performed by the main memory for any one lock are totally ordered; that is, for any two actions performed by the main memory on the same lock, one action precedes the other."
2. [p. 403] "Each lock or unlock action is performed jointly by some thread and the main memory."

Rule 5.1 supports our view of independent lock master agents. The actions on a single lock are ordered linearly, but actions on different locks may occur concurrently.

JLS rules 6 Let T be a thread and L be a lock.

1. [p. 406] "A lock action by T on L may occur only if, for every thread S other than T, the number of preceding unlock actions by S on L equals the number of preceding lock actions by S on L."
2. [p. 406] "An unlock action by thread T on lock L may occur only if the number of preceding unlock actions by T on L is strictly less than the number of preceding lock actions by T on L."

Rule 6.1 ensures that a thread's hold on a lock is exclusive, and Rule 6.2 ensures that for every unlock action there is a unique preceding lock action. But note that the rules allow several lock actions on the same lock to occur in succession, with no intervening unlock action. We find it convenient to think of a thread as building up a number of *claims* on a lock. A lock action adds a claim, and an unlock action takes one away. Rule 6.1 states that only one thread may

have a positive number of claims on the lock; furthermore, a thread may acquire multiple claims on a lock and surrenders the lock when and only when it has given up all claims. Rule 6.2 states that a thread may not release a claim on a lock it does not hold; the number of claims it has on a lock cannot be negative.

These rules ensure that the synchronization mechanism restricts synchronized code regions to one thread at a time. At most one thread at a time may have a positive number of claims on a given lock, and so if several threads need to lock the same lock, they must compete for it. Only the one with a positive number of claims on the lock is allowed into its synchronized region; the others must wait.

The function claims returns the number of claims a given thread has on a given lock. The function synchAction returns the action that the thread is requesting (if any).

A thread may only continue the execution of its program if it is active; *i.e.*, not synchronizing. We change the thread module accordingly, by guarding the rule *Execute program* with the term active?(Self).

term active?(t): undef?(synchAction(t))

To the rule *Execute program*, we add the options *Synch* and *Create object*. We add rules for threads' synchronization actions with locks. A thread synchronizes with a lock for either a lock action or an unlock action.

rule *Synch on ℓ to perform act*: synchAction(Self) := (act, ℓ)

rule *Synch*: **choose** *obj*: Object
choose *act*: LockActionType
Synch on lock(obj) to perform act

The rule *Create object* creates an object and associates it with a new lock. We also modify *Create thread* to associate each new thread with a new lock.

rule *Create object*: **extend** Object **with** *obj*
extend Lock **with** ℓ
lock(*obj*) := ℓ

The following JLS rules place some guarantees on the contents of a thread's working memory before and after synchronization actions.

JLS rules 7 [p. 407] "Let T be any thread, let V be any variable, and let L be any lock."

1. "Between an assign action by T on V and a subsequent unlock action by T on L, a store action by T on V must intervene; moreover, the write action corresponding to that store must precede the unlock action, as seen by main memory."
2. "Between a lock action by T on L and a subsequent use or store action by T on a variable V, an assign or load action on V must intervene; moreover, if it is a load action, then the read action corresponding to that load action must follow the lock action, as seen by main memory."

If a thread t issues an assign action A on a variable v and then issues an unlock action Ul on a lock ℓ, Rule 7.1 postulates that t issues a store action S on v (with corresponding write action W) such that $A < S < W < Ul$. Note that ℓ and v are independent; for an unlock action on a particular lock ℓ, Rule 7.1 applies to *all* variables v. Thus when a thread releases a claim on a lock, it is ensured that all its assigned values have been stored out and written to the main memory.

If a thread t issues a lock action Lk on a lock ℓ and then issues a use or store action US on a variable v, Rule 7.2 postulates that either t issues an assign action A on v such that $Lk < A < US$, or t issues a load action L (with corresponding read action R) on v such that $Lk < R < L < US$. For a lock action on a particular lock ℓ, Rule 7.2 applies to all variables v. So this rule ensures that by the time a thread acquires a claim on a lock, all the values cached in its working memory have been flushed out to main memory. When a thread acquires a claim on a lock, it acts as if its working memory is empty. Only values assigned or loaded in after the lock action may be used, and only values assigned after the lock action may be stored out.

The conditions imposed by these rules have some ramifications for earlier rules. A read message issued before a lock action cannot be loaded in after the lock action, but Rule 2.1 dictates that any such message must be loaded in. Therefore, all pending read messages must be loaded in before a lock action. Also, a value assigned before a lock action cannot be stored out after the lock action, but Rule 3.2 dictates that it must be stored out. Therefore, all assigned values must be stored out before a lock action.

The actions of a lock master consist of granting a claim to a lock and taking one away.

```
module Lock:  choose among
                  choose t: Thread: lockOK?(t)
                      Lock for t
                  choose t: Thread: unlockOK?(t)
                      Unlock for t

rule Lock for t:  claims(t, Self) := claims(t, Self) + 1
                  synchAction(t) := undef
                  do-forall v: Var: usableValue?(t, v)
                      usableValue?(t, v) := false

rule Unlock for t:  claims(t, Self) := claims(t, Self) - 1
                    synchAction(t) := undef
```

The terms lockOK? and unlockOK? determine whether a given action is allowed. A lock action for a thread t on a lock ℓ is allowed only if t is synchronizing for a lock action on ℓ. Rule 6.1 allows such a lock action only if all threads other than t have no claims on ℓ. Rules 3.2 and 7.2 require that all previously assigned values have been stored out. Rules 2.1, 2.3 and 7.2 require that all access messa-

ges have been loaded in, and Rules 4.1 and 7.2 require that all access messages from volatile variables have been used.

term lockOK?(t):
synchAction(t) = (lock, Self) and ($(\forall t'$: Thread: $t' \neq t$) claims(t', Self) = 0)
and (not $\exists v$: Var) freshAssign?(t, v) or ($\exists m$: AccMsg(v, t))
and (not $\exists v$: VolVar) usableValue?(t, v)

term unlockOK?(t):
synchAction(t) = (unlock, Self) and claims(t, Self) > 0
and (not $\exists v$: Var) freshAssign?(t, v) or ($\exists m$: UpdMsg(t, v))

5 Prescient Store Actions

A standard store action sends a value to the main memory from the working memory, where it was put by a preceding assign action. Under certain circumstances, Java allows an optimization in which a store action may *precede* its corresponding assign action, allowing the value to be transmitted to the main memory sooner than otherwise possible. This type of store action is called *prescient*, as it must be known ahead of time what the value of the assign action will be.

A prescient store action differs from a normal store action in that the value it sends to the main memory is not the current contents of working memory, but rather some fresh value. We call the following assign action *retroactive*, since the value it puts into working memory is not a fresh value, but rather the value of the preceding prescient store action. We find it natural to speak of prescient store actions and retroactive assign actions as distinct from regular store and assign actions.

We modify the ASM to model prescient store actions. If a given thread has issued a prescient store action on a given variable, without a corresponding retroactive assign action, the function presStoreVal returns the value of the prescient store.

We define rules for the new actions of prescient store and retroactive assign. Rule 8.4 disallows a store action on v. We also modify the rules *WM-EE transfer* and *WM-MM transfer*, adding prescient store and retroactive assign as options.

rule *Store presciently to* v: **choose** v: Var: presStoreOK?(Self, v)
freshAssign?(Self, v) := false
choose val: Value
presStoreVal(Self, v) := val
extend UpdMsg(Self, v) **with** m
val(m) := val

rule *Assign retroactively to* v: usableValue?(Self, v) := true
presStoreVal(Self, v) := undef
workingValue(Self, v) := presStoreVal(Self, v)

Prescient store actions must obey the following rules set out in the JLS:

JLS rules 8 [p. 408] "Suppose that a store by [a thread] T of [a non-volatile variable] V would follow a particular assign by T of V according to the rules of the previous sections, with no intervening load or assign by T of V. The special rule allows the store action to instead occur before the assign action, if the following restrictions are obeyed:

1. If the store action occurs, the assign is bound to occur.
2. No lock action intervenes between the relocated store and the assign.
3. No load of V intervenes between the relocated store and the assign.
4. No other store of V intervenes between the relocated store and the assign.
5. The store action sends to the main memory the value that the assign action will put into the working memory of thread T."

If a thread t performs a prescient store action PS on a variable v, there is a retroactive assign action $RA > PS$ by t on v such that RA assigns the value stored out at PS (Rules 8.1 and 8.5). Rules 8.1 and 8.5 ensure that the value t sends to main memory via a prescient store action ends up in t's working memory via a following retroactive assign action.

Furthermore, there is no action Act by t, where Act is a lock action or a load or store action on v, such that $PS < Act < RA$ (Rules 8.2-4). These rules ensure that relocating t's store action on v (*i.e.*, making it prescient) does not affect program behavior, in the sense that the effects of a prescient store action on t's working memory and the main memory are no different from those of the corresponding non-prescient store action.

The terms presStoreOK? and retroAssignOK? determine whether a prescient store action or retroactive assign action is allowed by the JLS rules, respectively. Rules 2.5 and 3.2 allow t to store to v presciently only if every AccMsg from v to t has been loaded in and there is no retroactive assign pending. Rule 8 restricts prescient store actions to volatile variables, and allows a retroactive assign action only if there is a preceding prescient store action without a corresponding retroactive assign action.

term presStoreOK?(v): not VolVar(v) and undef?(presStoreVal(Self, v))
and (not $\exists m$: AccMsg(v, Self))

term retroAssignOK?(v): def?(presStoreVal(Self, v))

Notice that t may issue a use action U on v between PS and RA. If the presciently stored value were to appear in t's working memory before t put it there at RA, t could end up using the presciently stored value prematurely. To prevent this, Rule 8.3 prohibits t from loading in from v between PS and RA. For the same reason, albeit less obviously, Rule 8.4 prohibits any lock action for t between PS and RA. This is needed because if a lock action for t were to occur and then t were to use v between PS and RA, Rule 7.2 would require a load action between PS and U, and such a load action is forbidden by Rule 8.3.

Relocating a store action makes sense only if there is no attempt to store between the relocated (prescient) store) action and its (retroactive) assign action. If t were to issue a store action S on v (with corresponding assign action A) between PS and RA, we would get the following undesirable result. S would follow PS, but the order of the corresponding assign actions would be different: RA would follow A. Thus the value stored through PS would be newer (*i.e.*, assigned more recently) than the value stored through the following action S. But Rule 2.5 would dictate that the newer value be overwritten in main memory by the older value. To prevent this, Rule 8.4 prohibits t from storing out a value of v between PS and RA.

Some of the rules introduced in previous sections (namely, 3.2–5 and 7.1–2) refer to store and assign actions and so must be modified to accommodate prescient store and retroactive assign actions. The rules as they appear in the JLS are not modified when prescient store actions are introduced.

We modify the terms lockOK? and loadOK? so that they enforce Rules 8.2–8.4. If a thread t has stored out presciently to v but has not yet performed the corresponding retroactive assign action, Rule 8.2 disallows any lock action for t, Rule 8.3 disallows a load action by t on v, and Rule 8.4 disallows any store action by t on v. To lockOK?(t), loadOK?(t) and storeOK?(t) we add the conjunct ($\forall v$: Var) undef?(presStoreVal(t, v)).

Rule 8.1 ensures that if a prescient store action on a variable occurs, a corresponding retroactive assign action will occur sometime in the future. As with previous rules, this rule can be flouted by delaying a retroactive assign action indefinitely. We restrict attention to runs that avoid this situation.

6 `Thread` Objects

We now consider how Java represents threads as objects. An object is an instance of a programmer-defined type called a *class*. Objects are created dynamically during a thread's computation. A class defines the state variables of its instances and the *methods* (operations) that may be invoked upon them. Each thread is represented by an object which is an instance of the class `Thread`. As there is a one-to-one correspondence between threads and their representative objects, we may identify a thread with its object. Information on a thread's state is held in its object. The methods defined in `Thread` allow a thread to access or modify its own state information and that of other threads.

We modify the ASM to model operations on threads. Since we identify a thread with its `Thread` instance, a member of the universe Thread is both a thread and a `Thread` instance. Note that `Thread` instances are members of both Thread and Object.

After a thread is created, it may be started by invoking the method `start` upon it. Once started, a thread is *alive* until it *stops*, which occurs when the thread's execution engine terminates or the method `stop` is invoked upon it. If `stop` is invoked on a thread, the thread *throws an exception*, signaling the occurrence of an unexpected event. Once a thread stops, it is no longer alive

and cannot be restarted. It is possible to stop a thread that has not started; if this happens, the thread will never become alive. It is not stated explicitly in the JLS what happens if `start` is invoked upon a thread that is already alive. However, the intent seems to be that the invoker of `start` throws an exception and nothing happens to the invokee [12].

We define functions started? and stopped? to represent the status of each thread. We also define rules *Start* and *Stop*, which update them appropriately.

The JLS rules impose some obligations on threads which they may not be able to fulfill after they have stopped. By Rules 2.1 and 2.3 (following the strict interpretation of "uniquely paired"), every access message must be loaded in. By Rule 4.1, every load action on a volatile variable must be followed by a corresponding use action, and by Rule 4.2, every assign action to a volatile variable must be followed by a corresponding store action. The JLS does not make it clear whether these obligations extend to threads that have stopped. Furthermore, it is not clear which choice is the more reasonable one. For instance, in some cases it may be desirable to have the last assign actions of a thread stored out, while in other cases it may be clearly undesirable; for example, consider a thread which is stopped because it is computing erroneous results. Officially, this is still an open issue [21]. For simplicity, we take the view that stopped threads are not obligated to do any further actions.

The JLS rules impose some obligations on variables which may not be fulfilled by the time the execution of all threads terminates; in particular, by Rule 2.4, every store message must be written. Thus it may still be necessary for variables to write some update messages in even after all threads have terminated.

A thread that is alive can be *suspended*, in which case it remains alive but its execution engine does nothing. A thread is suspended when some thread (possibly itself) invokes the method `suspend` upon it. A suspended thread *resumes* (becomes unsuspended) when some other thread invokes the method `resume` upon its `Thread` instance. It may be useful to suspend a thread when it cannot make progress by itself, freeing resources for other threads that can make progress. Can a suspended thread issue load or store actions? Officially, this is another unresolved issue [21]. Here we choose the more permissive interpretation, allowing suspended threads to issue read and load actions.

We add the function suspended? to represent the suspended status of each thread. We also define rules *Suspend* and *Resume*, which update this function.

A thread may be marked as a *daemon*. Such threads are intended to execute only in conjunction with non-daemon threads, performing "housekeeping" actions which assist non-daemon threads, but no actions useful in themselves. For this reason, once all non-daemon threads have stopped, execution of all threads halts. A program starts with a single non-daemon thread. A new thread starts with the daemon status of the thread that creates it, but its status can be changed via the `setDaemon` method.

To our ASM we add the function daemon?, which determines a thread's current daemon status, and the rule *Set daemon status*, which updates this function. We also modify the rule *Create thread*. A thread is added to the Object

universe and is initialized as unstarted, unstopped and unsuspended. It inherits the daemon status of the thread that created it.

We define a rule *Invoke thread method*, which chooses among the options *Start*, *Stop*, *Suspend* and *Set daemon status*. We modify the *Execute program* rule, adding the option *Invoke thread method*. To active? we add the conjuncts not suspended?(t) and undef?(synchAction(t)). We also modify the Thread module, guarding *Execute program* with alive?(Self) and not HaltAllThreads?

We define the term alive? to represent the running status of a thread. Also, the term HaltAllThreads? determines whether all non-daemon threads have terminated.

term alive?(t): started?(t) and not stopped?(t)

term HaltAllThreads?: ($\forall t$: Thread: alive?(t)) daemon?(t)

We modify the modules for variables and locks. If all non-daemon threads have terminated, the only action a variable may take is to write in update messages, so in the Var module we guard the read option with not HaltAllThreads?. There are no actions for locks to take once all non-daemon threads have terminated, so in the Lock module we guard the lock and unlock options with the term not HaltAllThreads?.

7 Waiting and Notification

While a thread holds a particular lock, other threads in need of that lock are not allowed to proceed. In certain cases, a thread holding a lock may reach a state from which it cannot progress by itself; in such a case, suspending the thread may not be a solution, as the thread would still hold the lock even when suspended. It would be desirable for the thread to give up control and release its locks temporarily, so that other threads may acquire them.

Java has built-in support for this, through the mechanisms of *waiting* and *notification*. If a thread has a claim on the lock of an object, it may release all its claims on the lock and disable itself by invoking the method `wait` of class `Object` on the object. Another thread may signal to the waiting thread that further progress is possible through the `notify` or `notifyAll` methods. In this way, a waiting thread may resume execution when it is possible, without repeatedly checking its state. Each object has a *wait set*, empty when the object is created, which contains all threads waiting for the lock on the object.

The method `wait` is invoked by a thread t on an object obj. The thread t must hold the lock of obj; if it does not, an exception is thrown. Let k be the number of claims t has on the lock of obj; then k unlock actions are performed, and t is added to the wait set of obj and disabled. When t is re-enabled, it attempts to regain k claims on the lock; it may need to compete with other threads to do this. Once the lock claims are restored, the `wait` method terminates.

We modify the ASM to model waiting and notification actions. A thread goes through several phases while it waits on a lock. First it synchronizes with the

lock to release all claims on the lock. Then it waits to be notified by another thread. Once it is notified, it synchronizes with the lock to regain all the claims it released. For a thread that is waiting on a lock, the function waitMode gives the current phase of the thread, the function waitLock gives the lock on which it is waiting, and claimsToRegain gives the number of claims that it has (temporarily) released.

We introduce a rule for a thread's actions while waiting on a lock. Waiting involves synchronization with the lock during the unlock and relock phases. Note that the waiting thread does not change its own mode from wait to relock; this is done by another thread, through a notification.

rule *Start to wait*: **choose** *obj*: Object: claims(Self, lock(*obj*)) > 0
    waitMode(Self) := unlock
    waitLock(Self) := lock(*obj*)
    claimsToRegain(Self) := claims(Self, lock(*obj*))

rule *Continue to wait*:
if waitMode(Self) = unlock **then**
  if claims(Self, waitLock(Self)) > 0 **then**
    Synch on waitLock(Self) to perform unlock
  else waitMode(Self) := wait
elseif waitMode(Self) = relock **then**
  if claims(Self, waitLock(Self)) < claimsToRegain(Self) **then**
    Synch on waitLock(Self) to perform lock
  else waitMode(Self) := undef

The method `notify` is also invoked by a thread t on an object *obj*. The thread t must hold the lock of *obj*; if it does not, an exception is thrown. If the wait set of *obj* is not empty, one thread is removed from it and enabled. The method `notifyAll` operates similarly but removes and enables all threads from the wait set. Neither method releases any of t's claims on the lock of *obj*, so there is no guarantee that the lock will be made available to the notified threads; this is left up to the programmer.

We introduce rules for notification. The method `notify` operates on a particular thread waiting on a particular lock; the method `notifyAll` operates on all threads waiting on a particular lock.

rule *Notify* t: waitMode(t) := relock

rule *Notify*:
choose *obj*: Object
  choose t: Thread: waitMode(t) = wait and waitLock(t) = lock(*obj*)
    Notify t

rule *NotifyAll*:
choose *obj*: Object
  do-forall t: Thread: waitMode(t) = wait and waitLock(t) = lock(*obj*)
    Notify t

When a thread stops, the method `notifyAll` is invoked upon its object. This allows a straightforward implementation of a technique called *thread joins*. Consider a scenario where a thread t' is expected to start executing only after another thread t has stopped. One way of implementing this is to have t' *join* with t by waiting until t has stopped before proceeding. Since `notifyAll` is invoked when a thread stops, t' may join with t by simply waiting on t's object. Since stopping a thread involves invoking `notifyAll` on its representative object, we change our ASM rule for stopping threads accordingly.

We point out a subtle issue. It is not clear from the JLS whether notification precedes stopping or *vice versa* when a thread stops. However, it seems reasonable that if stopping and notification do not occur as a single action, then notification must precede stopping (at least in the case where a thread stops itself). This leaves open the possibility that other threads may be notified of the thread's stop action before the thread has actually stopped, particularly if the notification process takes a relatively long time.

Threads may execute waiting or notification actions. Execution of a Java program does not proceed while a thread is waiting, so to the term active? we add the conjunct waitMode(t) $\neq$ wait. To the rule *Execute program*, we add the options *Start to wait*, *Notify* and *NotifyAll*. We also add the guard def?(waitMode(Self)); if this evaluates to true, then the rule *Continue to wait* fires; otherwise, one of the other options is chosen.

8 Conclusion

The Java concurrency model is currently in a state of flux. Researchers are proposing modifications to the model, to allow common compiler optimizations and programming practices that are currently prohibited by the model [19]. It is our belief that a satisfactory successor to the current model must start with a firm foundation. The specification of the model must not be subject to multiple interpretations, as is the description in the JLS. It is equally important that the specification be accessible to programmers who wish to use concurrency. While multithreaded programming is inherently complicated, the method of documentation should not exacerbate the problem.

We feel that this work has several things to offer the Java community. Our alternative view of Java concurrency brings to light some issues that might otherwise remain hidden in the JLS description. As the ASM model is mathematically precise, it can stand as an unambiguous cross-platform standard for the language. Our account of Java concurrency has an imperative flavor that is familiar to programmers, yet we are not restricted to a purely imperative approach; we are free to use a declarative style wherever it is more natural to do so. By implementing the concurrency model (albeit in a completely abstract way) we can readily see how the various constraints of the JLS definition interact with one another. This is much less obvious if constraints are presented in isolation, as they are in the JLS. Finally, programmers interested in multithreaded Java can explore the model by using the ASMGofer prototype. We believe that ASMs are a useful specification tool for both the current and future models of Java concurrency.

References

1. I. Attali, D. Caromel, and M. Russo. A formal executable semantics for Java. In *Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java*, 1998.
2. E. Börger. Why use Evolving Algebras for hardware and software engineering? In *Proceedings of SOFSEM*, 1995.
3. E. Börger and W. Schulte. A programmer friendly modular definition of the semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*. Springer, 1998.
4. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. ¿From sequential to multi-threaded Java: An event-based operational semantics. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, pages 75–90. Springer, 1997.
5. E. Coscia and G. Reggio. A proposal for a semantics of a subset of multi-threaded "good" Java programs. In *Proceedings of the OOPSLA '98 Workshop on the Formal Underpinnings of Java*, 1998.
6. A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations for Java memory behavior. Technion/CS Technical Report CS0922, 1997.
7. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
8. Y. Gurevich. Evolving Algebras: an attempt to discover semantics. In G. Rozenberg and A. Salomå, editors, *Current Trends in Theoretical Computer Science*, pages 266–292. World Scientific, 1993.
9. Y. Gurevich. Evolving Algebras 1993: Lipari guide. In E. Börger (editor), *Specification and Validation Methods*, Oxford University Press, 1995, 9–36.
10. Y. Gurevich. May 1997 draft of the ASM guide. Available at [22], 1997.
11. Y. Gurevich, W. Schulte and C. Wallace. Investigating Java concurrency using Abstract State Machines. Technical Report 2000-04, Department of Computer & Information Sciences, University of Delaware, 1999.
12. C. Horstmann and G. Cornell. *Core Java 1.1, volume II: Advanced Features*. Sun Microsystems Press, 1998.
13. Sun Microsystems. Java technology home page. `http://java.sun.com/`.
14. A. Jolin. Java's atomic assignment: The key to simpler data access across threads. *Java Report* 3(8), 27–36, 1998.
15. D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1997.
16. M. MacBeth, K. McGuigan and P. Hatcher. Executing Java threads in parallel in a distributed memory environment. In *Proceedings of IBM CASCON*, 1998.
17. S. Oaks and H. Wong. *Java Threads*. O'Reilly and Associates, 1997.
18. G. Plotkin. Structural Operational Semantics (Lecture notes). Technical Report DAIMI FN-19, Aarhus University, 1981.
19. W. Pugh. Fixing the Java memory model. In *Proceedings of ACM Java Grande*, 1999.
20. Joachim Schmid. ASMGofer home page. `http://www.tydo.de/AsmGofer/`.
21. G. Steele. Personal communication.
22. Univ. of Michigan. ASM home page. `http://www.eecs.umich.edu/gasm/`.
23. Univ. of Paderborn. ASM home page. `http://www.uni-paderborn.de/cs/asm/`.

Verifying Compilers and ASMs
or
ASMs for Uniform Description of Multistep Transformations

Gerhard Goos and Wolf Zimmermann

Universität Karlsruhe
Fakultät für Informatik
{ggoos,zimmer}@ipd.info.uni-karlsruhe.de

Abstract. A verifying compiler ensures that the compiled code is always correct but the compiler may also terminate with an error mesage and then fails to generate code. We argue that with respect to compiler correctness this is the best possible result which can be achieved in practice. Such a compiler may even include unverified code provided the results of such code can be proven correct independently from how they are generated. We then show how abstract state machines (ASMs) can be used to uniformly describe the dynamic semantics of the programs being compiled across the various intermediate transformation steps occurring within a compiler. Besides being a convenient tool for describing dynamic semantics the fact that we do not have to switch between different descriptional methods is found to be extremely useful.

1 Introduction

Many statistics attribute more than 50% of all software failures to problems in requirement engineering whereas programming errors account for a much smaller percentage, cf. [23,49]. But what if everything in software development was correct except that errors were introduced during compilation? This case is not as infrequent as some people think, cf. [52,37,12,13,14]. Especially programs often show different behavior with and without optimization. The generated code should exactly behave like the source program; this is of utmost importance especially in safety-critical applications. The requirement can only be ensured by a compiler which verifiably produces correct code. But despite many efforts, cf. e. g. the efforts for validating Ada-compilers, no compiler on the market for a realistic programming language such as Ada, C, C++ or Java is fulfilling this requirement. It is therefore no surprise that safety surveillance institutions such as the "Technische Überwachungsvereine" (TÜV) in Germany often do not accept software written in high-level languages but instead check the resulting assembly code and require the use of non-optimizing compilers.

McCarthy and Painter, [36], were first considering correctness of compilation but for arithmetic expressions only. Many people dealt with the problem

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 177–202, 2000.

thereafter as discussed in Sect. 9 but nobody succeeded in producing a correct compiler for a realistic programming language with a typical machine language as target. In our opinion two main causes have lead to this:

- The problem was considered in an idealistic, i. e. mathematical setting in which common data types such as integer and float were considered like integers and reals in mathematics, i. e. the range and precision limitations on computers were ignored. In the same vain storage limitations at compile- or run-time and other types of resource limitations were ignored.
- The formal methods chosen for describing the source and target language and of the intermediate languages in the compiler made the treatment of realistic languages too difficult. As a consequence attention was restricted to relatively small programming languages disregarding the complexities and dark sides of realistic languages. For the same reason code generation was mostly restricted to stack machines using the data types of the source language. Also, the software architecture for correct compilers in past efforts was specially adapted to the verification needs.

In the present contribution we first discuss the notion of correctness. We arrive at the conclusion that the main goal should be the correctness of the target program produced, not primarily the correctness of the compiler itself. This insight allows for reusing the results of compiler research of the past decades and for using conventional compiler architectures. This discussion is closely related to the problem how to deal with resource limitations of all kinds. We then introduce Gurevich's abstract state machines (ASMs) and show that this method is a suitable means for formalizing the (dynamic) semantic of programming languages on all levels of representation. The advantages include the facts that we do not have to specially adapt the compiler architecture to verification needs, and that we do not have to switch between different formalisms for describing semantics during the compilation process.

2 Correctness of Compilation

We consider a pair (PL,ML) consisting of an imperative or object-oriented programming language PL such as Ada, C, C++, Sather-K, [28], or Java and a microprocessor such as the DEC Alpha represented by its machine language ML. For simplicity we restrict ourselves to sequential programming languages.

The translation of a source program π in PL to a target program $\pi' = \mathcal{C}(\pi)$ in ML is certainly correct if π' shows the same behavior as π, i. e. on all inputs π' is producing the same outputs as π.

A state of a computation is called *observable* if it is the initial or final state or, if the operation leading into this state is an input or output operation. Observable states are the only states representing an effect noticeable in the environment of a computation.[1] If we consider states q, q' of the execution of π and π' respectively as sets of state variables then the requirement *same behavior* asks for

[1] The programmer or compiler writer may, at his discretion, designate further states, e. g. procedure entries or exits, as observable.

identifying corresponding state variables in q and q' holding the same values in corresponding observable states. We represent this requirement by a relation ρ between states of π' and π.

Correctness of translation then requires that a sequence $q_0, q_1, \ldots, q_i, \ldots$ of observable states in the execution of π translates into a sequence $q'_0, q'_1, \ldots, q'_i, \ldots$ of observable states of π' and $\rho(q_i) = q'_i$ holds for corresponding states.

The behavior of π may be indeterministic: Fig. 1 shows the acyclic graph of possible execution paths for the program

```
do true -> x := 1
[] true -> x := 0
od
```

in DIJKSTRA's language of guarded commands. The implementation may choose

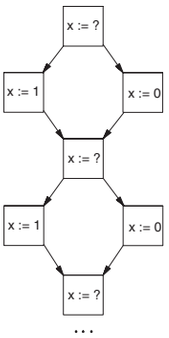


Fig. 1. Execution paths for an indeterministic program

an arbitrary path:

```
do true -> x := 1
od
```

is a correct translation of this program. For this reason we can only require that each sequence of observable states of the translated program $\pi' = \mathcal{C}(\pi)$ possesses a corresponding state sequence of the source program π; the converse is not true.

Ideally we want to preserve the termination behavior of the source program. For reasons discussed in the next section this is not possible in a realistic environment: we must admit that the target program π' is prematurely terminating due to excessive resource demands. Thus, during compilation we cannot preserve total but only partial correctness. Altogether this leads to the following definition of correct translation:

Definition 1. *A target program π' is a correct translation of a source program π, $\pi' = \mathcal{C}(\pi)$, if for all inputs one of the following conditions applies:*

- *To each sequence* $\mathcal{S}' = q'_0, q'_1, \ldots, q'_k, \ldots$ *of observable states of* π' *there is a sequence* $\mathcal{S} = q_0, q_1, \ldots, q_k, \ldots$ *of observable states of* π *with* $q'_i = \rho(q_i)$ *for* $i = 0, \ldots, k, \ldots$ *If* $\mathcal{S}'$ *regularly terminates after* k *steps then also* $\mathcal{S}$.
- *To each sequence* $\mathcal{S}' = q'_0, q'_1, \ldots, q'_k, q'_{k+1}$ *of observable states of* π' *terminating with an error message due to a resource violation in state* q'_{k+1} *there is a sequence* $\mathcal{S} = q_0, q_1, \ldots, q_k, \ldots$ *of observable states of* π *with* $q'_i = \rho(q_i)$ *for* $i = 0, \ldots, k$.

In both cases the state sequences must be complete with respect to observability.

3 Dealing with Resource Limitations

Mathematically speaking the length of nearly all programs exceeds k lines where k is an arbitrarily large number, e. g. the number of atoms in the universe. A compiler $\mathcal{C}$ running on a computer in the real world cannot cope with such programs. It must instead be prepared for refusing source programs because the compilation would demand more resources, e. g. storage, than is available. We could even argue that

```
begin
 print("program not compilable due to excessive resource requirements")
end
```

is a correct compiler for arbitrary pairs (PL,ML) without regard how those languages are defined.

Thus, correct compilation is primarily an engineering issue: we want to produce this error message for as few programs as possible and deliver a correct target program for the remaining source programs. But requiring that a compiler is delivering correct target programs for *all* source programs is unrealistic.

For becoming realistic we must restrict our goal: We do not want to construct *correct* compilers but are instead interested in *verifying compilers*: they ensure the correctness of the target program in case they deliver one; and, of course, we want to make the set of correctly translated programs as large as possible.

For similar reasons a compiler cannot be kept responsible if at runtime of the generated program an integer multiplication overflows, or if the number and size of objects to be allocated exceeds certain bounds, or if the operating system on the target machine cannot cope with the number of simultaneously accessed external devices and communication lines, etc. In a realistic environment the programmer of the source program must take precautions against these problems, not the compiler writer. For all such reasons we must allow not only the compiler but also the generated target program to prematurely terminate with an error message due to violation of resource limitations.

If the processor running the compiler or the target program is too slow to achieve the desired result in acceptable time we also have a violation of resource limitations. In practice, when real-time constraints are exceeded, it is infeasible to decide whether the programmer was demanding too much or the compiler generated bad code or the target processor was too slow. We thus do not count execution speed of the target program amongst the correctness criteria.

4 Compiler Architecture

Let us assume that we have subdivided a compilation $\mathcal{C}\colon \pi \to \pi'$ into a sequence of two or more translation steps $\mathcal{C}'\colon \pi \to \pi''$, $\mathcal{C}''\colon \pi'' \to \pi'$, and that each step yields correct results according to definition 1. Then, as long as there are no resource violations, each observable state q' of π' has a corresponding observable state q'' with $q' = \rho''(q'')$ and in turn there is an observable state q with $q'' = \rho'(q)$ and hence $q' = \rho''(\rho'(q))$ with appropriately defined relations ρ' and ρ''. If we define $\rho(q) = \rho''(\rho'(q))$ then definition 1 is therefore also fulfilled for the sequence $\mathcal{C} = \mathcal{C}';\mathcal{C}''$ of these steps.

We term this property *vertical compositionality* of translation steps. It allows the introduction of an arbitrary number of intermediate program representations $\pi'', \ldots$ into the compilation process. If each individual compilation step is correct then also the sequential composition of these steps. Traditional compi-

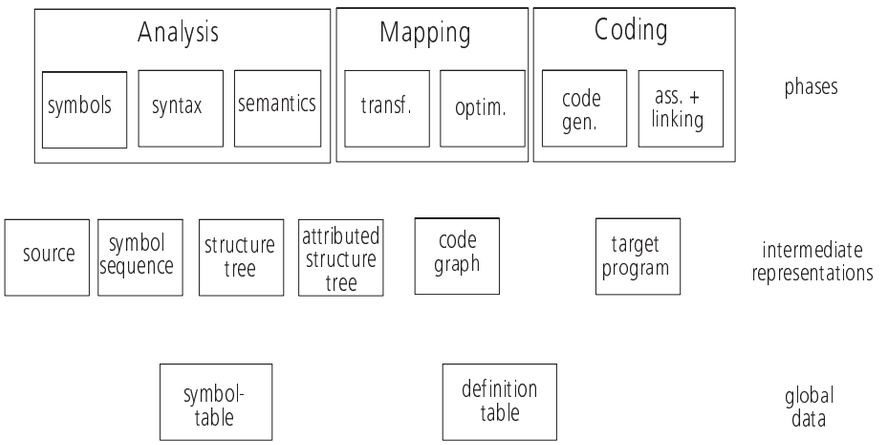


Fig. 2. Compiler architecture

ler architecture is using vertical compositionality for decomposing compilations into a sequence of simpler steps each of which can be dealt with by appropriate methods and tools, cf. Fig. 2 and e. g. [54,39].

Actually, the first row of Fig. 2 shows a decomposition of a compiler into modules only, not necessarily into sequential steps; however, locally for each element of an intermediate representation the phases follow each other sequentially from left to right. Conceptually it is thus justified to consider the phases as sequential steps although the actual merging in time may be much more complex.

The semantics of programming languages and thus the behavior of program executions is usually defined by attaching meaning to certain phrases of a program as given by the (context-free) grammar of the language. The meaning of

larger phrases and the program as a whole is then derived by substituting the semantics of subphrases into the semantics of larger ones. This property is called *horizontal compositionality* (of language elements).

There may be syntactic and semantic limitations for the admissibility of subphrases. A program is termed statically correct if these conditions are fulfilled. Compilers check these conditions during semantic analysis and refuse to generate code for incorrect programs. When discussing verifying compilers we may restrict attention to statically correct programs.

Horizontal compositionality permits to deal with one language element at a time during the vertically decomposed translation steps. It thus adds to the structuring of the compiler. The method is, of course, not applicable before we have determined the phrase structure and during optimization steps which on purpose consider several language elements together.

This raises the question how we can certify the correctness of the abstract syntax tree for a given program text. The requirement *same behavior* does not help since the dynamic behavior is only defined for the phrases visible in the syntax tree whereas the program text is only serving as a vehicle for deriving this tree. Similarly, as soon as we have decided which instruction sequences correctly represent the intended behavior of phrases in the source program we have established the correctness of the target program. The remaining assembly and linking phase only decides about the proper encoding of the resulting instructions but does not change the semantic meaning. So to say, the assembly phase acts like the inverse of the transformation source text → syntax tree, but on a different level of abstraction.

Thus, basically only the mapping phase in Fig. 2 is concerned with the semantic correctness of compilation. The remaining phases deal with decoding and encoding the structures which carry semantic meaning. Especially the translation process seen as a mapping between source and target semantics is confined to the mapping phase only. This insight has been ignored for a long time in the discussion about correct compilation.

For lexical analysis, parsers, code selectors and other phases routinely generators are used for deriving the corresponding compiler phase. Verifying such phases would require to verify the correctness of the corresponding generators, a very heavy and burdensome task. Previous work on correct compilation thus relied on hand-writing these phases and verifying the hand-written code. In the next section we show that there is an easier way of dealing with this problem when we relax the requirements and only request that (for certain source programs) the output of a phase is correct instead of requesting that the code of the compiler phase is producing correct results under all circumstances. Under this condition we can use unverified generators and other tools as in traditional compilers and nevertheless trust the resulting code.

5 Verifying Compilers: The Case for Program Checking

Program checking was originally introduced for algorithms by Blum and Kannan, [1]. We present it here as in [26] for verifying the outputs of systems such as compilers or parts thereof.

Let $\mathcal{C}$ be a program implementing a function $f\colon I \to O$ with precondition $P(x)$ and postcondition $Q(x, \mathcal{C}(x))$ on input x. Let $checker(x, y) : Bool$ be a function that returns the value of $Q(x, y)$. Consider the program

```
function C'(x : I) : O
   y := C(x);
   if checker(x, y) then return y
   else abort;
end;
```

and assume that $\mathcal{C}'(x)$ does not abort. Then the result $Q(x, \mathcal{C}'(x))$ holds if *checker* is correctly implemented. Thus, if $\mathcal{C}'$ does not abort then the result $y = \mathcal{C}(x)$ fulfills its postcondition no matter how it was computed. In particular, the partial correctness of $\mathcal{C}'$ does not depend on the partial correctness of $\mathcal{C}$. Therefore we only need to verify *checker* but not $\mathcal{C}$ for getting a verified result.

In practice the verification of *checker* is often much simpler than the verification of $\mathcal{C}$. This is particularly true if $\mathcal{C}$ contains a lot of optimizations which, from the correctness point of view, are irrelevant as long as they maintain some simple correctness conditions. E. g. the register allocator within a compiler is correct as long as the values assigned to the same register have non-overlapping lifetimes; an optimizing register allocator must maintain this condition while it is devoting most of its effort to avoiding spill code as far as possible; its quality in this respect does not influence its correctness.

Table 1. Lines of program code to verify for a program-checked IS front-end

	C/Sather-K		Binary Prog.
	Lines	Byte	Byte
Generators COCKTAIL	110.000	2.4 MB	1.2 MB
Generated C Code Impl. IS-Frontend	22.000	600 KB	300 KB
Checker (Sather-K)	500 (Parser) +100 (Compare) +700 (AST)	14 KB 3 KB 30 KB	200 KB

Table 2. Lines of program code to verify for example back-end

	Modula/Sather-K		Binary Prog.
	Lines	Byte	Byte
Generator BEG (Modula)	35.000	2 MB	1.1 MB
Generated C Code Impl. MIS Code-Selection	18.000	400 KB	500 KB
Checker (Sather-K)	500 (Parser) +300 (Checker) +400 (MIS)		200 KB
Industrial: ANDF ⇒ ST9	140.000	6.4 MB	3.5 MB

We applied this approach to a compiler front-end for a C subset IS, [19, 32]. Details can be found in [31]. Tab. 1 shows the results. Tab. 2 shows similar results for the code selection phase using the back-end generator BEG, [22,21]. The checker has been written in the object-oriented language SATHER-K, [28]; a subset of SATHER-K comparable to JAVA will be the first realistic language for which we expect to present a verifying compiler. As can be seen in both cases the verification effort for the checker is by orders of magnitude lower than for the generator or the generated compiler phase.

6 What Has to Be Verified

Program checking only verifies that a result y obeys its postcondition $Q(x, y)$. Nevertheless the result could be wrong because the postcondition Q does not ensure the required properties of y. For a verifying compiler we must distinguish several proof obligations:

1. Verification of the *compiling specification*: The compiling specification describes the correspondence of source and target language in formal terms. Source and target languages are usually given by informal descriptions, e. g. an ISO-norm for the source language and processor descriptions by hardware manufacturers.
2. Verification of the *compiler specification*: This specification introduces the various compiler phases, intermediate representations and data structures, e. g. as in Fig. 2. Formally it must be verified that vertical composition of the compiler phases leads to the mapping given by the compiling specification.
3. Verification of the *compiler implementation*: A proof is required that the various parts of the compiler architecture are correctly implemented according to the compiler specification.

4. Verification of the *compiler encoding*: No matter whether the compiler is implemented as a program in a high-level language or directly in assembly code hardware only accepts bit sequences as inputs. Thus we have to ensure that the symbolic code is properly encoded by bit sequences as required by the hardware.

The compiling specification can only be verified by hand; verification tools cannot be applied since the original descriptions cannot be explored by formal means.

In practice compiling and compiler specification are usually combined. Then the specifications underlying the compiler phases such as the regular expression describing the lexemes, the context-free grammar, etc. must be verified by hand. This combination reveals that the specification of the mapping phase relating notions and concepts of the source and the target language is the crucial part of all these specifications.

As far as theorem provers can be applied the Verifix project relies on the use of PVS, cf. [20]. Only the verification of the implementation can be (partially) replaced by program checking.

Verification of the encoding can be achieved by program checking based on the specification of the mapping of symbolic assembly code to binary code. In principle also the results of the linking phase and of the system loader must be ckecked; this is presently not part of our project.

Compiler phases and program checkers are programs themselves and their correctness depends on the correctness of the compilers used to translate them. To break the cycle that a verifying compiler assumes the existence of another verifying compiler, a suitable bootstrap process is discussed in [27].

7 Abstract State Machines

So far we have discussed what has to be done for achieving a verifying compiler and argued that such a compiler could be built along the guidelines of traditional compiler architecture. Underlying was the assumption that we have formal description methods at our disposal by which we can formalize the requirement *same behavior* and verify that it is maintained during a compilation.

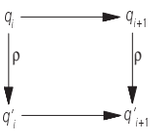


Fig. 3. Corresponding observable states

The requirement asks for establishing the relation ρ between corresponding observable states as in Def. 1; the diagram Fig. 3 must commute. To this end

we must look into the details of the representation of states and how they are dynamically transformed by executing statements and instructions respectively. According to our experience abstract state machines, [29,], provide adequate means to formally handle this problem.

An *Abstract State Machine* (ASM) is a tuple $(\Sigma, \Phi_{\text{Init}}, \Phi_{\text{Inv}}, \mathit{Trans})$ where Σ is a signature, Φ_{Init} and Φ_{Inv} are sets of Σ-formulas (the *initial* and *invariant conditions*), and *Trans* is a finite set of *transition rules*. A transition rule is a pair $(\varphi, \mathit{Updates})$, denoted by

if φ **then** *Updates*

where φ is a Σ-formula and *Updates* is a set of pairs $t_1 := t_2$ of Σ-terms, called *updates*. The set of *states* is the set $\text{Alg}(\Sigma)$ of Σ-algebras that satisfy the Σ-formulas in Φ_{Inv}, i. e. q is a model for Φ_{Inv} in the sense of logic, $q \models \Phi_{\text{Inv}}$; $[\![\cdot]\!]_q$ denotes the interpretation function of symbols of Σ in Σ-algebra q. A state is *initial* iff $q \models \Phi_{\text{Init}}$.

Remark 1. This definition of ASMs slightly differs from the usual definition. Instead of defining the states to be Σ-algebras, we define them as sets of Σ-algebras that satisfy a set of formulas. By this modification we can prove desired properties of ASMs by logical calculi more easily.

The *state transition relation* $\rightarrow$ is based on transition rules. Intuitively, a transition rule $(\varphi, \text{Updates})$ *fires* if its condition φ is satisfied. Formally speaking the condition is true in state q iff $q \models \varphi$. The updates $t_1 := t_2$ of the rule are then executed and change the interpretation of some symbols in the state. A state q is *final* iff no rule fires in q.

Let $\mathit{Exec}(q)$ be the set of updates executed in state q. $\mathit{Exec}(q)$ is *consistent* iff for all[2] $f(t_1, \ldots, t_k) := t,\ f(t'_1, \ldots, t'_k) := t'$, $q \models t_1 \doteq t'_1 \wedge \cdots \wedge t_k \doteq t'_k \Rightarrow t \doteq t'$ holds. Then the state transition relation $q \rightarrow \bar{q}$ is defined by

$$[\![f]\!]_{\bar{q}}(a_1, \ldots, a_n) = \begin{cases} [\![t]\!]_q & \exists\ f(t_1, \ldots, t_n) := t \in \mathit{Exec}(q) \\ & \text{with } [\![t_i]\!]_q = a_i, i = 1, \ldots, n \\ [\![f]\!]_q(a_1, \ldots, a_n) & \text{otherwise} \end{cases}$$

for each k-ary function symbol f of Σ iff $\mathit{Exec}(q)$ is consistent, $\bar{q} \models \Phi_{\text{Inv}}$ and $[\![T]\!]_q = [\![T]\!]_{\bar{q}}$ for all sorts of Σ (i.e. the interpretation of sorts is not changed by the state transition).

8 Compilers as Multistep Transformation Systems

In this section, we focus on proving the correctness of the compiling specification, i. e. that the observable behavior of a program is preserved during compilation. We assume that the dynamic semantics of a program is given by an ASM (not only its observable behavior).

For sequential languages and processors a state transition $q \rightarrow \bar{q}$ can be split into two parts:

[2] $t_1 \doteq t_2$ denotes a Σ-equation.

1. determine the flow of control, i. e. the next state $\bar{q}$;
2. execute the remaining updates occurring during this transition.

The next statement or instruction (after a conditional jump) $\bar{q}$ possibly depends on a Boolean expression with a value already known in state q but unknown during compilation. $\bar{q}$ is represented in language and processor descriptions by distinguishing a state variable *next task* indicating the next state. As long as we solely consider the flow of control, its refinements and its translation we can abstract from the details of ASMs and concentrate on this state variable only. This leads to the notion of state transition systems (STS) as defined in Sect. 8.1.

State transition systems deal with dynamic sequencing of states. When a compiler transforms the state sequence then it also transforms the sequence of observable states. Thus, the observable states to be preserved can be defined within state transition systems although the proof that the behavior is actually preserved requires details only available in the full ASM descriptions.

The principle of constructing the operational semantics of a program by horizontal composition as explained in Sect. 4 leads to compositions of state transition systems from "smaller" ones defined in Sect. 8.2. Refinements of composed state transitions systems can be obtained from refinements of their components.

8.1 State Transition Systems

Formally, a *state transition system* (STS) is a triple $\mathcal{S} = (Q, I, \rightarrow)$ where Q is a set of states, $I \subseteq Q$ is a set of *initial* states and $\rightarrow \subseteq Q \times Q$ is a transition relation. A state $q \in Q$ is *final* iff there is no $q' \in Q'$ with $q \rightarrow q'$. A *run* qq *of* $\mathcal{S}$ is a finite or infinite sequence of states $\langle q_0, q_1, q_2, \ldots \rangle$ such that $q_0 \in I$ and $q_{i-1} \rightarrow q_i$ for all $0 < i$. A run is *complete* iff it is infinite or its last state is final. A *sub-run* of a run qq is a subsequence of qq. A sequence of states is a sub-run of $\mathcal{S}$ iff it is a sub-run of some run of $\mathcal{S}$. A STS is *trivial* iff $\rightarrow = \emptyset$ and *basic* iff $\rightarrow \subseteq I \times (Q \setminus I)$. We only consider non-trivial STSs. Every run of a non-trivial basic STS consists exactly of one state transition.

An ASM $\mathcal{A} = (\Sigma, \Phi_{\text{Init}}, \Phi_{\text{Inv}}, \mathit{Trans})$ defines a STS $\mathcal{S}_\mathcal{A} = (Q, I, \rightarrow)$ with

- $Q = \{q \in \text{Alg}(\Sigma) \mid q \models \Phi_{\text{Inv}}\}$
- $I = \{q \in \text{Alg}(\Sigma) \mid q \models \Phi_{\text{Inv}} \cup \Phi_{\text{Init}}\}$, and
- $\rightarrow$ is the transition relation of $\mathcal{A}$.

Let $\mathcal{S} = (Q, I, \rightarrow)$, $\mathcal{S}' = (Q', I', \rightarrow')$ be STSs and $\phi : Q \rightarrow Q'$ be a partial function with domain $\text{DOM}(\phi)$. Compilers transform programs frequently by refining them: a single state transition of $\mathcal{S}'$ is replaced by a sub-run of a more detailed state transition system $\mathcal{S}$. $\mathcal{S}$ *ϕ-refines* $\mathcal{S}'$ iff

(i) $I \subseteq \text{DOM}(\phi)$,
(ii) $\phi(I) \subseteq I'$,
(iii) $\phi(q_0) \rightarrow' \phi(q_n)$ for every sub-run $\langle q_0, \ldots, q_n \rangle$ of $\mathcal{S}$ with $q_0, q_n \in \text{DOM}(\phi)$, $\phi(q_0) \neq \phi(q_n)$, and $q_1, \ldots, q_{n-1} \notin \text{DOM}(\phi)$, cf. Fig. 4.

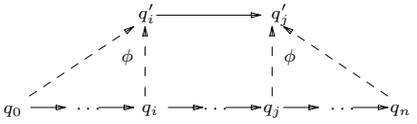


Fig. 4. Refinement and abstraction of STSs

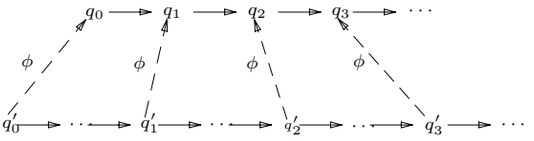


Fig. 5. Horizontal decomposition

Conversely $\mathcal{S}'$ is called a *ϕ-abstraction* of $\mathcal{S}$. The refinement (abstraction) is 1:1 iff ϕ is injective and $\phi(q') \to \phi(\bar{q}')$ for every $q' \to' \bar{q}'$. The refinement (abstraction) is *complete* iff ϕ is surjective. The refinement (abstraction) is *total* iff ϕ is a total function with the additional properties that $\phi(q)$ is final in $\mathcal{S}'$ if q is final in $\mathcal{S}$ and there is no infinite sub-run $\langle q_i : i \in \mathbb{N}\rangle$ with $\phi(q_i)$ final for any $i \in \mathbb{N}$. $\mathcal{S}$ and $\mathcal{S}'$ are *isomorphic* iff $\mathcal{S}'$ ϕ-refines $\mathcal{S}$ for a total bijective function $\phi : Q \to Q'$.

Remark 2. For final states q of $\mathcal{S}$ it is not ensured that $q \in \mathrm{DOM}(\phi)$ and $\phi(q)$ is final in $\mathcal{S}'$. It seems desirable that properties analogous to (i) and (ii) hold for final states. However, for constructing correct compilers the definition above is exactly what is needed: A final state $q \notin \mathrm{DOM}(\phi)$ represents an abortion of an execution due to violation of resource limitations.

Remark 3. The semantics of a program will usually define many intermediate states between two observable states. We may abstract the ASM describing such a program to a STS or, conversely, we may consider the ASM description as a refinement of a STS which only contains observable states. Then it is this abstraction, not the details of the original ASM, which must be preserved during compilation. The abstraction function ϕ is thus a special case of the more general relation ρ between states of two STSs.

If $\mathcal{S}'$ refines a STS $\mathcal{S}$ then every run of $\mathcal{S}'$ induces a run of $\mathcal{S}$, cf. Fig. 5:

Theorem 1. *Let $\mathcal{S} = (Q, I, \to)$ and $\mathcal{S}'$ be two* STS*s, and $\phi : Q \to Q'$ a partial function. $\mathcal{S}$ ϕ-refines $\mathcal{S}'$ iff each run of $\mathcal{S}'$ is a sequence of sub-runs $\langle q'_0, \ldots, q'_i\rangle$, $\langle q'_i, \ldots, q'_j\rangle, \cdots$ such that for each sub-run $\langle q'_i, \ldots, q'_j\rangle$ one of the following conditions holds:*

- *There is a sub-run $\langle q_i, \ldots, q_k, \ldots, q_j\rangle$ of $\mathcal{S}$ and $\phi(q_k) = q'_k$ for $k = i, \ldots, j$.*

– $j = i + 1$ *and there is a sub-run* $\langle q^{(0)}, \ldots, q^{(m)} \rangle$ *of* $\mathcal{S}$ *with* $\phi(q^{(0)}) = q'_i$, $\phi(q^{(m)}) = q'_j = q'_{i+1}$ *and* $q^{(k)} \notin \mathrm{DOM}(\phi)$ *for* $0 < k < m$.

The refinement relation on STS is transitive:

Theorem 2 (Stepwise Refinement). *Let* $\mathcal{S}_i = (Q_i, I_i, \rightarrow_i)$, $i = 1, 2, 3$, *be three state transition systems. If* $\mathcal{S}_3$ ϕ_2*-refines* $\mathcal{S}_2$ *and* $\mathcal{S}_2$ ϕ_1*-refines* $\mathcal{S}_1$, *then* $\mathcal{S}_3$ $\phi_1 \circ \phi_2$*-refines* $\mathcal{S}_1$. $\mathcal{S}_3$ *is a total refinement of* $\mathcal{S}_1$ *if both refinements are total.*

We are interested in these stepwise refinements since they are used in horizontal composition and by translation steps which act like macro-substitutions. But compilers based on pure macro-expansion produce inefficient code in particular for RISC-processors. Stepwise refinement for example would forbid optimizations that change the order of state transitions such as instruction scheduling and code motion.

To cope with this problem we replace in Theorem 2 the abstraction function ϕ_2 by a more general relation $\rho \subseteq Q_3 \times Q_2$ but still request that $\mathcal{S}_3$ is a refinement of $\mathcal{S}_1$. Then the relation $\phi' = \phi_1 \circ \rho$ must be a partial function. Furthermore conditions (i)–(iii) for refinements must be ensured by ϕ'.

For condition (i) we require $I_3 \subseteq \mathrm{DOM}(\rho) = \{q^{(3)} \in Q_3 \mid \exists q^{(2)} \in Q_2 . q^{(3)} \ \rho \ q^{(2)}\}$ and $\rho(I_3) \subseteq I_2$. Since $I_2 \subseteq \mathrm{DOM}(\phi_1)$ we then have $\rho(I_3) \subseteq \mathrm{DOM}(\phi_1)$ and thus $I_3 \subseteq \mathrm{DOM}(\phi')$.

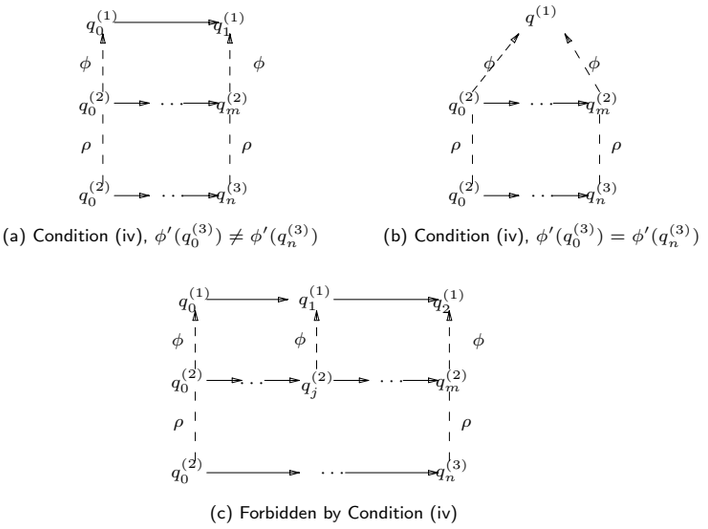
$\rho(I_3) \subseteq I_2$ also implies condition (ii): since $\mathcal{S}_2$ ϕ_1-refines $\mathcal{S}_1$ we have $\phi_1(I_2) \subseteq I_1$. Hence $\phi'(I_3) \subseteq \phi_1(I_2) \subseteq I_1$.

For condition (iii) let $\langle q_0^{(3)}, \ldots, q_n^{(3)} \rangle$ be a sub-run of $\mathcal{S}_3$ with $q_0^{(3)}, q_n^{(3)} \in \mathrm{DOM}(\phi')$, $q_1^{(3)}, \ldots, q_{n-1}^{(3)} \notin \mathrm{DOM}(\phi')$, and $\phi'(q_0^{(3)}) \neq \phi'(q_n^{(3)})$. Then $\phi'(q_0^{(3)}) \rightarrow_1 \phi'(q_n^{(3)})$ must hold. Obviously, there must be states $q_0^{(2)} \in \rho(q_0^{(3)}) \cap \mathrm{DOM}(\phi)$ and $q_m^{(2)} \in \rho(q_n^{(3)}) \cap \mathrm{DOM}(\phi)$ such that $\phi(q_0^{(2)}) \neq \phi(q_m^{(2)})$. Hence, we must require $q_0^{(2)} \rightarrow_2^+ q_m^{(2)}$ for at least one pair of these states.[3] Furthermore, there must be at least one sub-run $\langle q_0^{(2)}, \ldots, q_m^{(2)} \rangle$ with $q_1^{(2)}, \ldots, q_{m-1}^{(2)} \notin \mathrm{DOM}(\phi)$; otherwise there would be at least two state transitions from $\phi'(q_0^{(3)})$ to $\phi'(q_n^{(3)})$, cf. Fig. 6(c). This property can be ensured by requiring $\mathrm{DOM}(\phi) \subseteq \mathrm{RAN}(\rho) = \{q^{(2)} \in Q_2 \mid \exists q^{(3)} \in Q_3 . q^{(2)} \ \rho \ q^{(3)}\}$. Together we have

Theorem 3 (Vertical Decomposition). *Let* $\mathcal{S}_i = (Q_i, I_i, \rightarrow_i)$, $i = 1, 2, 3$, *be three state transition systems such that* $\mathcal{S}_2$ ϕ*-refines* $\mathcal{S}_1$ *for a partial function* $\phi : Q_2 \rightarrow Q_1$. *Let* $\rho \subseteq Q_3 \times Q_2$ *be a relation with the following properties:*

(i) $\phi' = \phi \circ \rho$ *is a partial function.*
(ii) $I_3 \subseteq \mathrm{DOM}(\rho)$ *and* $\rho(I_3) \subseteq I_2$
(iii) $\mathrm{DOM}(\phi) \subseteq \mathrm{RAN}(\rho)$.
(iv) *For every sub-run* $\langle q_0^{(3)}, \ldots, q_n^{(3)} \rangle$ *with* $q_0^{(3)}, q_n^{(3)} \in \mathrm{DOM}(\phi')$ *there are states* $q_0^{(2)} \in \rho(q_0^{(3)}) \cap \mathrm{DOM}(\phi)$ *and* $q_m^{(2)} \in \rho(q_n^{(3)}) \cap \mathrm{DOM}(\phi)$ *such that* $q_0^{(2)} \rightarrow_2^+ q_m^{(2)}$.

Then $\mathcal{S}_3$ ϕ'*-refines* $\mathcal{S}$ *for a function* $\phi' : Q_3 \rightarrow Q_1$.



(a) Condition (iv), $\phi'(q_0^{(3)}) \neq \phi'(q_n^{(3)})$ (b) Condition (iv), $\phi'(q_0^{(3)}) = \phi'(q_n^{(3)})$

(c) Forbidden by Condition (iv)

Fig. 6. Vertical decomposition

We say that $\mathcal{S}_3$ *ρ-simulates* $\mathcal{S}_2$ if $\rho \subseteq Q_3 \times Q_2$ under the conditions of theorem 3. ρ-simulation allows for optimizations that reorder execution.

Remark 4. Condition (iv) considers all sub-runs beginning and ending with states and is more general than argued above. However, every such sub-run can be decomposed into sub-runs that do not contain intermediate states in the domain of ϕ'. If $\phi'(q_0^{(3)}) \neq \phi'(q_n^{(3)})$ then condition (i) implies that there are states $q_0^{(2)} \in \rho(q_0^{(3)}) \cap \mathrm{DOM}(\phi)$ and $q_m^{(2)} \in \rho(q_n^{(3)}) \cap \mathrm{DOM}(\phi)$ such that $q_0^{(2)} \neq q_m^{(2)}$. Fig. 6(a) and (b) illustrate condition (iv) for the case $q_0^{(2)} \neq q_m^{(2)}$ and $q_0^{(2)} = q_m^{(2)}$ respectively.

Mapping the data of a high-level language program to bits and bytes is a data-refinement. When using ASMs we change signatures and implement the algebras by other algebras. Given a partial injective mapping $\phi : \mathrm{Alg}(\Sigma', \Phi'_{\mathrm{Inv}}) \to \mathrm{Alg}(\Sigma, \Phi_{\mathrm{Inv}})$ then an ASM $\mathcal{A}' = (\Sigma', \Phi'_{\mathrm{Init}}, \Phi'_{\mathrm{Inv}}, \mathit{Trans}')$ is a *ϕ-data-refinement* of $\mathcal{A} = (\Sigma, \Phi_{\mathrm{Init}}, \Phi_{\mathrm{Inv}}, \mathit{Trans})$ iff $\mathcal{A}'$ 1:1-refines $\mathcal{A}$ and for each transition rule $(\varphi', \mathit{Updates}')$ there is exactly one transition rule $(\varphi, \mathit{Updates})$ such that $q' \models \varphi'$ implies $\phi(q') \models \varphi$.

Example 1. Assume the operational semantics a programming language is defined by an ASM. Assume the state of a program in a given programming language PL contains data objects allocated in an environment (the procedure stack) and on a heap. A compiler for PL must map these objects to the concepts of the

[3] As usual, $\to^+$ denotes the transitive closure of a binary relation $\to$.

target machine, i. e. to byte adressable memory. Besides the given operational semantics $\mathcal{A}$ of PL, given by an ASM, we need a second operational semantics $\mathcal{A}'$ for PL using the concepts of the target machine. $\mathcal{A}'$ is a data-refinement of $\mathcal{A}$. But given the limited memory of the target machine $\mathcal{A}'$ contains bounds beyond which it will refuse to allocate additional objects.

Refinement functions ϕ are not required to be total. The example exhibits the reason: the stack and the heap of a high-level programming language are unlimited, whereas the memory of the target machine is limited. Hence, a run of $\mathcal{A}'$ may end in an exceptional state q_e where no more data can be stored. Obviously, $q_e \notin \mathrm{DOM}(\phi)$.

8.2 Programs as State Transition Systems: Composition of STSs

By using horizontal compositionality state transition systems for statements and expressions are combined for obtaining the STS for sequential or conditional execution of statements, for loops, etc. We view the composed STSs and its components as graphs and describe the composition as a graph building process. The building blocks are state transition systems as in Fig. 7 with initial and final state (sets) I and T. Basic STSs are drawn by ovals instead of rectangles.

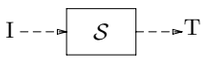


Fig. 7. Basic Components for Composition

For two STSs $\mathcal{S}_1, \mathcal{S}_2$ an edge $\mathcal{S}_1 \rightarrow \mathcal{S}_2$ as in Fig. 8(a) means sequential execution of $\mathcal{S}_1$ and $\mathcal{S}_2$. It requires that the postcondition T_1 of $\mathcal{S}_1$ implies the precondition I_2 of $\mathcal{S}_2$, i.e. each final state of $\mathcal{S}_1$ is an initial state of $\mathcal{S}_2$. $\mathcal{S}_1$ could also have several final states with different postconditions to which we may attach different successors; then $\mathcal{S}_1$ acts as a selector; by arriving at a certain final state it selects amongst its successors. In this case, we annotate the corresponding edge with the set of final states. In order to avoid new non-determinism, the sets associated with the out-edges of an STS must be pairwise-disjoint. Furthermore, an STS can only be entered at an initial state and left at a final state. Finally, the initial states of a composed STS must be defined. This is modeled by the special vertex I: an initial state of a component $\mathcal{S}$ is initial in the composed STS iff there is an edge $(I, \mathcal{S})$.

Then a directed labeled graph $G = (V, E)$ with $V = \{\mathcal{S}_1, \ldots, \mathcal{S}_k \cup \{I\}$ with initial node I (precondition) is a *composition* of the k STSs $\mathcal{S}_1 = (Q_1, I_1, \rightarrow_1), \ldots, \mathcal{S}_k = (Q_k, I_k, \rightarrow_k)$ if each edge $(\mathcal{S}_i, \mathcal{S}_j)$ is associated with a subset $F_{i,j}$ of final states F_i of $\mathcal{S}_i$, respectively, and the following conditions are satisfied:

(i) There exists at least one index i and an edge $(I, \mathcal{S}_i)$. (There is a component $\mathcal{S}_i$ to begin with.)

(ii) $F_{i,j} \subseteq F_i \cap I_j$. (For every edge $(\mathcal{S}_i, \mathcal{S}_j)$, each state of $F_{i,j}$ is final in $\mathcal{S}_i$ and initial in $\mathcal{S}_j$.)

(iii) $F_{i,j} \cap F_{i,k} = \emptyset$ for every $j \neq k$. (Each final state of $\mathcal{S}_i$ unambigously determines its successor.)

(iv) A common state $q \in Q_i \cap Q_j$ of two components $\mathcal{S}_i, \mathcal{S}_j$ either belongs to $F_{i,j} \cap F_{j,i}$, or there is a $\mathcal{S}_k$ such that $q \in F_{i,k} \cap F_{j,k}$. (If two components have common states then either there is an edge between the two components and the state is in the set associated with the edge, or they have a common successor $\mathcal{S}_k$ and the state is in the sets associated with both edges $(\mathcal{S}_i, \mathcal{S}_k)$ and $(\mathcal{S}_j, \mathcal{S}_k)$).

Let $\mathcal{S}_{j_1}, \ldots, \mathcal{S}_{j_l}$ be the successors of I in G. The STS $\mathcal{S}_G = (Q_1 \cup \cdots \cup Q_k, I_{j_1} \cup \cdots \cup I_{j_l}, \rightarrow_1 \cup \cdots \cup \rightarrow_k)$ is the STS *described by G*. For convenience, we assume $F_{i,j} = \emptyset$ if there is no edge $(\mathcal{S}_i, \mathcal{S}_j)$.

Condition (iii) ensures that no new non-determinism is introduced by composition, condition (iv) forbids undesired "jumps" out of a STS.

If the STSs are abstractions from ASMs then the conditions (ii)–(iv) about the state sets $F_{i,j}$ translate into formulae about pre- and postconditions of ASMs:

(ii') $\Phi^{(i,j)}_{\text{final}} \models \Phi^{(i)}_{\text{Inv}} \cup \Phi^{(j)}_{\text{Inv}} \cup \Phi^{(i)}_{\text{final}} \cup \Phi^{(j)}_{\text{Init}}$ and $\Phi^{(i)}_{\text{Inv}} \cup \Phi^{(j)}_{\text{Inv}} \models \Phi^{(i,j)}_{\text{final}}$ for each edge $(\mathcal{A}_i, \mathcal{A}_j)$.

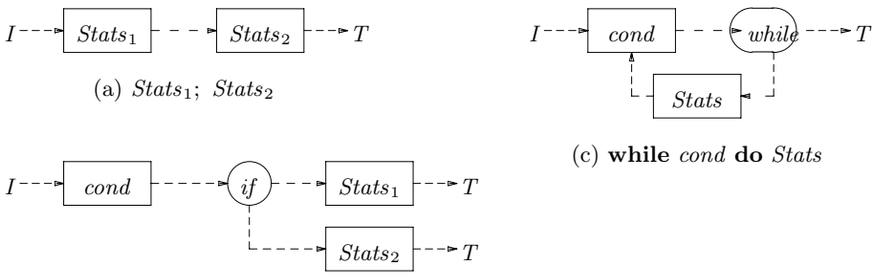
(iii') For every component $\mathcal{A}_i$ with successors $\mathcal{A}_h, \mathcal{A}_j$, the set $\Phi^{(i,h)}_{\text{final}} \cup \Phi^{(i,j)}_{\text{final}}$ is inconsistent.

(iv') For every pair $\mathcal{A}_i, \mathcal{A}_j$ and every formula φ such that $\Phi^{(i)}_{\text{Inv}} \models \varphi$ and $\Phi^{(j)}_{\text{Inv}} \models \varphi$ either $\Phi^{(i,j)}_{\text{final}} \models \varphi$, $\Phi^{(j,i)}_{\text{final}} \models \varphi$, or $\mathcal{A}_i$ and $\mathcal{A}_j$ have a common successor $\mathcal{A}_h$ such that $\Phi^{(i,h)}_{\text{final}} \models \varphi$ and $\Phi^{(j,h)}_{\text{final}} \models \varphi$.

Remark 5. A state q of the STS $\mathcal{S}_G$ described by the composition G is final iff it is a final state of some component $\mathcal{S}_i$ and there is no out-edge $(\mathcal{S}_i, \mathcal{S}_j)$ with $q \in F_{i,j}$. We may make this property explicit by adding a new special vertex T, an edge $(\mathcal{S}_i, T)$ and associate with q the set $F_{i,T}$ of those final states of $\mathcal{S}_i$ which are final in $\mathcal{S}_G$.

Example 2. The STS of the sequential composition $Stats_1; Stats_2$ is defined by the sequential composition of the STS of $Stats_1$ and $Stats_2$, cf. Fig. 8(a). A conditional statement **if** *cond* **then** $Stats_1$ **else** $Stats_2$ is defined as the sequential composition of the STS for *cond*, a single state transition *if*, and the disjoint union of the STSs for $Stats_1$ and $Stats_2$, cf. Fig. 8(b). A loop **while** *cond* **do** *Stats* is defined as in Fig. 8(c) by a cyclic graph consisting of the STS for *cond*, a single state transition *while*, and the STS for *Stat*.

Obviously, this example only defines the possible control-flow of a program viewed as sequences of possible state transitions. But how e. g. the while node decides amongst its possible successors is yet unspecified. With ASMs this detail can be added:



(a) $Stats_1$; $Stats_2$

(b) **if** *cond* **then** $Stats_1$ **else** $Stats_2$

(c) **while** *cond* **do** *Stats*

Fig. 8. STS representing control-flow

Example 3. Fig. 9 shows the signature of ASMs for a while-language. The signature contains sorts *TASK* representing the sort of tasks (or instructions) belonging to various instruction classes. Tasks are represented by constant symbols, i. e. nullary functions of the algebraic signature Σ, for each instruction occurring in a program. Similarly, variables are represented by a constant for each occurring variable. Fig. 10 shows state transition rules for instruction classes. These are used for the basic STSs defined in Fig. 8. For example, the oval vertex *if* represents an ASM with the transition rule for *IF* and $\Phi_{\text{Init}} = \{CT \doteq if_j\}$ for a $j \in \{1, \ldots, k_1\}$. The function *NT* is a partial function. A last task is final. The conditions (i), (ii), (iv) require that each ASM in the composition has different tasks. Let $\mathcal{A} = (\Sigma, \Phi_{\text{Init}}, \Phi_{\text{Inv}}, \mathit{Trans})$ be a non-basic ASM used in the composition. Then, $\Phi_{\text{Inv}} = \{CT \doteq task_1 \vee \cdots CT \doteq task_k\}$ where $task_1, \ldots, task_k$ are a set of tasks, some of which are initial in successor ASMs. Φ_{Init} and Φ_{final} have the same form, e. g. Φ_{Init} specify a disjunction of equations $CT \doteq task_i$ that are initial. An edge $(\mathcal{A}, \mathcal{A}')$ also associated with a disjunction of equations $CT \doteq task$ where each of these equations occur in Φ_{final} as well as in Φ'_{Init}.

8.3 Montages

During compilation a program is represented by an abstract syntax tree. The elements of such trees carry semantic meaning and the meaning of a program is composed from the meaning of these elements. The abstract syntax "normalizes" the concrete syntax in several respects: it no longer cares about representational details such as use of keywords or rules for operator precedence etc. These details are encoded either in node names of subtrees or in the structure of an expression tree. Also selection rules such as

```
statement ::= assignment | procedure call | loop | ...
```

which only enumerate the possible alternatives, are avoided: whenever a statement is allowed then one of the alternatives is directly inserted in the abstract

$\Sigma \equiv$
sorts $INT, VAR, TASK$
subsorts $DECISION,\ ASSIGN,\ EXPR < TASK,\ DES, PLUS, CONST < EXPR$
$WHILE, IF < DECISION$

functions $\ldots, -2, -1, 0, 1, 2, \ldots :\rightarrow INT$
$x_1, x_2, \ldots, x_{k_0} :\rightarrow VAR$ where $x_1, x_2, \ldots, x_{k_0}$ are all identifiers of a program
$if_1, \ldots, if_{k_1} :\rightarrow IF$
$while_1, \ldots, while_{k_2} :\rightarrow WHILE$
$assign_1, \ldots, assign_{k_3} :\rightarrow ASSIGN$
$des_1, \ldots, des_{k_4} :\rightarrow DES$
$plus_1, \ldots, plus_{k_5} :\rightarrow PLUS$
$const_1, \ldots, const_{k_2} :\rightarrow CONST$

$content$:	VAR	$\rightarrow INT$	the memory
CT :		$\rightarrow TASK$	the task pointer
$value$:	$EXPR$	$\rightarrow INT$	the value of an expression
$+$:	$INT \times INT$	$\rightarrow INT$	
id :	DES	$\rightarrow VAR$	variable of a designator
$const$:	$CONST$	$\rightarrow INT$	value of a constant expression
NT :	$TASK$	$\rightarrow TASK$	the next task
TT :	$DECISION$	$\rightarrow TASK$	alternative decision
$dest$:	$ASSIGN$	$\rightarrow DES$	destination of an assignment
src :	$ASSIGN$	$\rightarrow EXPR$	source of an assignment
$cond$:	$DECISION$	$\rightarrow EXPR$	condition for a decision
lop, rop :	$PLUS$	$\rightarrow EXPR$	left and right operand

Fig. 9. Signature of ASMs for a While-Language

syntax tree.[4] Thus, the tree only contains leaves and so-called *composition productions* composing a language element from smaller units (subtrees). Every tree element has a semantic meaning which influences program execution.

Semantic analysis is concerned with evaluating context conditions i. e. name analysis, type checking/inference, and operator identification, and checking their consistency; it works on symbolic information and does not care about the dynamic interpretation of a program: whether an operation named "integer addition" really means an addition (not a subtraction) is of no concern. Even the question which algebraic laws are applicable is uninteresting during this phase. Traditionally semantic analysis is described by an attribute grammar, based on abstract syntax.

[4] Of course, given the non-orthogonal design of many languages the selection may depend on context conditions which must be checked during semantic analysis.

if CT **is** $IF \wedge \neg value(cond(CT) \doteq 0$ **then**
  $CT := TT(CT)$
if CT **is** $WHILE \wedge \neg value(cond(CT) \doteq 0$ **then**
  $CT := TT(CT)$
if CT **is** $ASSIGN$ **then**
  $content(id(dest(CT))) := value(src(CT))$
  $CT := NT(CT)$
if CT **is** DES **then**
  $value(CT) := content(id(CT))$
  $CT := NT(CT)$
if CT **is** $IF \wedge value(cond(CT)) \doteq 0$ **then**
  $CT := NT(CT)$
if CT **is** $WHILE \wedge value(cond(CT)) \doteq 0$ **then**
  $CT := NT(CT)$
if CT **is** $PLUS$ **then**
  $value(CT) := value(lop(CT)) + value(rop(CT))$
  $CT := NT(CT)$
if CT **is** $CONST$ **then**
  $value(CT) := const(CT))$
  $CT := NT(CT)$

Fig. 10. Transition Rules for a While-Language

Only when we enter the mapping phase of a compiler we get concerned with the actual (dynamic) semantics of a program: We need to know the control and data flow and certain information about the dynamic interpretation of operations. The required static information can again be represented by attribute grammars. The dynamic semantics must be correctly transformed by the mapping phase; its rôle is that of invariants which must be preserved.

This description can be represented for each production by a variant of KUTTER'S *montages*, [35]. Montages were developed for graphically describing the dynamic semantics of source languages by help of ASMs. We use it here for specifying semantic analysis and the mapping phase of a compiler. The main differences to KUTTER's monateges are that our context-free grammar defines abstract, not concrete syntax, and the static semantics is described by attribute grammars (not by ASMs).

A *montage* in this sense is a tuple consisting of a composition production $p : X_0 ::= X_1 \cdots X_k$, a control- and data-flow graph G_p, attribution rules R, conditions C_p, and a set of transition rules $Trans_p$. The graph G_p consists of four classes of vertices and edges. A *task vertex* is graphically represented by an ellipse. There is at most one task vertex representing the start of execution of the montage; this vertex must be labeled with X_0. *Nonterminal vertices* are labeled with one of the nonterminals of the RHS of p. I and T represent *initial* and *terminal* vertices, respectively.

A *control-flow* edge is drawn dashed, a *data-flow* edge is drawn solid. The origin of named edges must be task vertices. The origin of a data-flow edge must

be a task vertex. Each out-edge of a data-flow edge must be named. The out-edges of a square vertex X_i are labeled with integers. If X_i defines n terminal sets, then X_i has n out-edges. The destination of named control-flow edges must be either task vertices or the set of initial tasks has exactly one element. Analogously, the destination of named data-flow edges must be either task vertices or the set of terminal task has exactly one element.

Remark 6. This definition does not exclude non-determinism. It requires that each desired non-determinism is *explicitly* specified.

Fig. 11 shows the montage for the **while**-loop. The decision whether the loop ends or is continued rests on the value of the loop condition. This dependence is indicated by a data-flow edge. There may be additional ways for exiting the loop if (as in C) the body contains `continue` or `break`-statements.

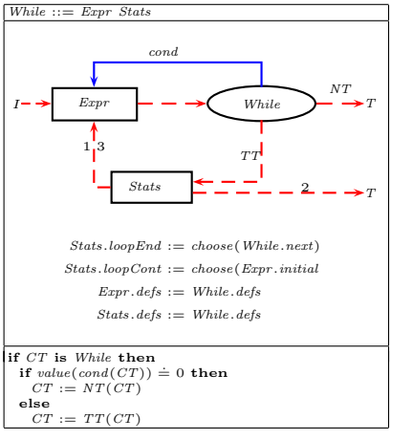


Fig. 11. A Montage for While-Loops

There are two views on the graph G_p: First it defines a set of attributes and attribution rules representing the task structure of programs. Together with the nodes and edges of the graph we have an ASM $\mathcal{A}_{SL}$ for the source language. Additionally we have a parametrized ASM $\mathcal{A}_{TL}$ for the target language when names and data-flow edges are removed from G_p. The mapping is implicitly specified as a conditional graph rewrite system. A montage thus definea a mapping from abstract syntax to control- and data-flow graphs.

The operational semantics of a program can be composed from the montages representing the involved language elements according to the structure of the syntax tree. Each square vertex represents a different subtree of the AST; conditions (ii)–(iv) of Section 8.2 are satisfied, since each value of CT specifies the currently executed ASM.

8.4 Horizontal Decomposition

The program transformations used by compilers transform the control- and data-flow graphs. We already discussed that the observable behaviour $\mathcal{O}$ of the program must be preserved, i.e. the ASM assigned to the transformed program must be a refinement of $\mathcal{O}$. The program transformations can be described by a finite set of local graph rewrite rules. Horizontal compositionality implies that the correctness of such rules can be proven locally. Since $\mathcal{O}$ is given by a (hierarchical) composition G of STSs, it must be ensured that composing refinements by the corresponding composition (i.e. using the same graph and replacing each component by a refinement) is a refinement of $\mathcal{O}$.

A composed STS (ASM) can be refined by refining its components. Let $G = (V, E)$ be a composition of $\mathcal{S}_1, \ldots, \mathcal{S}_k$, $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$ be $\phi_1, \ldots, \phi_k$-refinements of $\mathcal{S}_1, \ldots, \mathcal{S}_k$, respectively, and $\phi = \phi_1 \cup \cdots \cup \phi_k$. Furthermore, let $G' = (V', E')$ be the graph obtained from G by replacing $\mathcal{S}_1, \ldots, \mathcal{S}_k$ by $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$, respectively, and associating the sets $F'_{i,j} = \phi^{-1}(F_{i,j})$ with edges $(\mathcal{S}'_i, \mathcal{S}'_j)$. We now investigate the requirements for G' being a composition of $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$ and the STS obtained by G' being a refinement of G.

For G' being a composition of $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$ the condition Section 8.2(i) is satisfied whereas conditions (ii)–(iv) could be violated. However, this can be cured by "renaming" states. We leave the formal proof of this to the reader. When using montages, such a renaming is not necessary since the task pointer CT unambigously defines the current ASM.

Let $\mathcal{S}, \mathcal{S}'$ be the STS described G, G' respectively. ϕ must be a partial function for $\mathcal{S}'$ being a ϕ-refinement of $\mathcal{S}$. This is only the case iff $\phi_i(q) = \phi_j(q)$ for $q \in \mathrm{DOM}(\phi_i) \cap \mathrm{DOM}(\phi_j)$. It is not hard to see that conditions (i) and (ii) for $\mathcal{S}'$ being a ϕ-refinement of $\mathcal{S}$ are satisfied. It remains to prove condition (iii). Obviously, each sub-run of a component $\mathcal{S}'_i$ is also a sub-run of $\mathcal{S}'$. Hence, the only sub-runs that may violate condition (iii) are sub-runs $\langle q'_0, \ldots, q'_n \rangle$ with $q'_0, q'_n \in \mathrm{DOM}(\phi)$, and $q'_1, \ldots, q'_{n-1} \notin \mathrm{DOM}(\phi)$ where q'_0 and q'_n strictly belong to the states of two different STSs. However, there are no such sub-runs because $F'_{i,j} \subseteq I'_j \subseteq \mathrm{DOM}(\phi_j) \subseteq \mathrm{DOM}(\phi)$ for all i, j.

Hence, we obtain the

Theorem 4. *Let G, $\mathcal{S}_1, \ldots, \mathcal{S}_k$, $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$, $G' = (V', E')$, ϕ, and $F'_{i,j}$ be defined as above. If G' is a composition of $\mathcal{S}'_1, \ldots, \mathcal{S}'_k$ and if $\phi_i(q) = \phi_j(q)$ for all i, j, $i \neq j$, $q \in Q'_i \cap Q'_j$ then the* STS *$\mathcal{S}'$ obtained from G' is a ϕ-refinement of the* STS *$\mathcal{S}$ obtained from G.*

Again, it is not hard to see that these conditions can be ensured by simply renaming the states. In the case of montages, only $\phi_i(q) = \phi_j(q)$ for $q \in Q_i \cap Q_j$ has to be proven. If no data are refined, the condition is always true. Hence, the compositional refinement of montages is easy to prove whereas data-refinements (e.g. memory mapping) require additional work.

8.5 Correct Compilations

A compilation from one language to another consists of two tasks: data mapping and operation mapping. In this subsection, we demonstrate the use of the above theory for showing the correctness of compilations. We use the ASMs $\mathcal{A}_{SL}$ and $\mathcal{A}_{TL}$ as given by montages for implicitly describing data and operation mapping. Our goal is to independently prove the correctness of data mappings and conditional graph rewrite rules.

Consider first the data mapping. With STS, we decomposed ASMs in a "behavioral" part and in a data part. A data mapping only maps the data part and keeps the behavioural part. In particular, it assigns a new semantics (by means of an ASM $\mathcal{A}'_{SL}$) to the source language using the concepts of the data part of the target language (e.g. implementing a runtime stack using an address-based memory). Since the behavioural part is kept, the correctness of this mapping can be shown by proving that $\mathcal{A}'_{SL}$ 1-1-refines $\mathcal{A}_{SL}$. In order to complete the correctness proof, we have to show according to Theorem 3 that $\mathcal{A}_{TL}$ ρ-simulates $\mathcal{A}'_{SL}$.

Theorem4 can be used to prove that $\mathcal{A}_{TL}$ ρ-simulates $\mathcal{A}'_{SL}$. Observe that ASMs are STSs and we now use the behavioural part of an ASM, i.e. its view as a STS. Theorem 4 allows to independently prove the correctness of any graph rewrite rule. The relation ρ relates the initial and final states of the left-hand side and the right-hand side of the graph rewrite rule, respectively.

If a graph rewrite rule has an application condition, the correctness proof can assume that this condition is satisfied. The compiler verifies the satisfaction of this condition using program checking (in general it checks a stronger condition because the application condition may be state dependent).

9 Related Work

Correctness of compilers was first considered in [36] but focused on the compilation of arithmetic expressions. Thereafter most people explored the potential of denotational semantics, e. g. [15,40,41,44,45,48,56], or of refinement calculi, e. g. [6,4,8,16,17,33,38,43], structural operational semantics, e. g. [18] and algebraic models, e. g. [50]. Other approaches use abstract state machines, e. g. [6, 4,8]. Most of these projects did not compile into machine language. Instead, they designed abstract machines, and compiled for interpreters of these abstract machines.

These semantics-based approaches lead to monolithic compilers, cf. [24,53]. They do neither allow for reuse of traditional compiler technology nor do they prepare for program reorganizations, as necessary for global optimization on the machine code level. E. g., expressions are usually refined into postfix form and then interpreted on a stack machine. The efficiency of the generated code is by magnitudes worse than that of other compilers and thus does not meet practical requirements, cf. [18,44]. Except [38], even projects which really generated machine language, e. g. [6,4,42,43], and ProCos, [33], chose the transputer,

i. e., a stack machine, as their target. [38] discusses compilation of a stack-based intermediate language *Piton* into an assembly language of the register-based processor FM9001. The correctness proofs of the transformations as well as their implementation (in ACL2 logic) are checked mechanically using the ACL2 interpreter. In contrast to our work, the compilation is a macro-expansion and the source programs must be terminating regularly.

The idea of program checking was originally applied to algorithms in [1] and continued in [2,3,57]. [26] discusses its application to constructing correct systems. [46,47] apply the idea to translating synchronous languages (SIGNAL, Lustre, Statecharts) to C-programs; however, their assumptions allow only for (reactive) source programs consisting of a single loop; the loop body must implement a function from inputs to outputs; only the loop body is checked.

Many languages have been described so far using abstract state machines, e. g. C [30], C++ [55], Prolog/WAM [8], Occam/Transputer [6,4], Java [11], Java Virtual Machine [9,10,58], APE100 [5], ARM2 [34], DEC-Alpha [25], DLX [7].

10 Conclusions

We have introduced a notion of compiler verification which remains feasible also in the presence of the unavoidable limitations of realistic harware and software. This notion together with the idea of program checking has laid the ground for a compiler architecture very similar to traditional architectures. We thus can also generate efficient target code comparable to the code quality of other compilers. Nevertheless constructing verifying compilers will remain a tedious task for the foreseeable future.

In this work state transition systems and abstract state machines have shown their usefulness as a unifying descriptional tool for specifying the dynamic semantics of the programs being compiled on all levels of internal representation within the compiler. ASMs have not only lead to easily understood and natural specifications. The main advantage is that we do not have to change horses in the middle of the river and can apply the same specification seamlessly to all transformation steps. The expressive power of ASMs is only partially used in this work: The number of cases in the proofs of many theorems can be considerably reduced by adhering to a common discipline of the ASM features being used.

Acknowledgments. We are grateful to Hans Langmaack, Friedrich W. von Henke, Axel Dold, Thilo Gaul, Wolfgang Goerigk, Andreas Heberle, Ulrich Hoffmann, Markus Müller-Olm, Holger Pfeifer, Harald Rueß and many students in Karlsruhe, Kiel and Ulm for their contributions to the Verifix project which made this paper possible. The Verifix project is supported by the Deutsche Forschungsgemeinschaft under contract numbers Go 323/3-2, He 2411/2-2, La 426/15-2.

References

1. M. Blum and S. Kannan. Program correctness checking ... and the design of programs that check their work. In *Proceedings 21st Symposium on Theory of Computing*, 1989.
2. M. Blum, M. Luby, and R. Rubinfeld. Self–testing/correcting with applications to numerical problems. In *Proceedings 22nd Symposium on Theory of Computing*, 1990.
3. Manuel Blum and Sampath Kannan. Designing programs that check their work. *Journal of the Association for Computing Machinery*, 42(1):269–291, January 1995.
4. Egon Boerger, Igor Durdanovic, and Dean Rosenzweig. Occam: Specification and Compiler Correctness.Part I: The Primary Model. In U. Montanari and E.-R. Olderog, editors, *Proc. Procomet'94 (IFIP TC2 Working Conference on Programming Concepts, Methods and Calculi)*. North-Holland, 1994.
5. E. Börger, G. Del Castillo, P. Glavan, and D. Rosenzweig. Towards a Mathematical Specification of the APE100 Architecture: the APESE Model. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 396–401, Elsevier, Amsterdam, the Netherlands, 1994.
6. E. Börger and I. Durdanovic. Correctness of compiling occam to transputer. *The Computer Journal*, 39(1):52–92, 1996.
7. E. Börger and S. Mazzanti. A Practical Method for Rigorously Controllable Hardware Design. In J.P. Bowen, M.B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *LNCS*, pages 151–187. Springer, 1997.
8. E. Börger and D. Rosenzweig. The WAM-definition and Compiler Correctness. Technical Report TR-14/92, Dip. di informatica, Univ. Pisa, Italy, 1992.
9. E. Börger and W. Schulte. A Modular Design for the Java VM architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, 1998.
10. E. Börger and W. Schulte. Defining the Java Virtual Machine as Platform for Provably Correct Java Compilation. In *23rd International Symposium on Mathematical Foundations of Computer Science*, LNCS. Springer, 1998. To appear.
11. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
12. Borland/Inprise. Official borland/inprise delphi-3 compiler bug list. http://www.borland.com/devsupport/delphi/fixes/3update/compiler.html, jul 1999. Delphi3 Compiler Bug List.
13. Borland/Inprise. Official borland/inprise delphi-4 compiler bug list. http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html, jul 1999. Delphi4 Compiler Bug List.
14. Borland/Inprise. Official borland/inprise delphi-5 compiler bug list. http://www.borland.com/devsupport/delphi/fixes/delphi5/compiler.html, jan 2000. Delphi5 Compiler Bug List.
15. D. F. Brown, H. Moura, and D. A. Watt. Actress: an action semantics directed compiler generator. In *Compiler Compilers 92*, volume 641 of *Lecture Notes in Computer Science*, 1992.
16. B. Buth, K.-H. Buth, M. Fränzle, B. v. Karger, Y. Lakhneche, H. Langmaack, and M. Müller-Olm. Provably correct compiler development and implementation. In U. Kastens and P. Pfahler, editors, *Compiler Construction*, volume 641 of *LNCS*. Springer-Verlag, 1992.

17. Bettina Buth and Markus Müller-Olm. Provably Correct Compiler Implementation. In *Tutorial Material – Formal Methods Europe '93*, pages 451–465, Denmark, April 1993. IFAD Odense Teknikum.
18. Stephan Diehl. *Semantics-Directed Generation of Compilers and Abstract Machines*. PhD thesis, University of the Saarland, Germany, 1996.
19. A. Dold, T. Gaul, W. Goerigk, G. Goos, A. Heberle, F. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Definition of the Language IS. Verifix Working Paper [Verifix/UKA/1], University of Karlsruhe/Kiel/Ulm, 1995.
20. A. Dold, T. Gaul, and W. Zimmermann. Mechanized verification of compiler back-ends. In B. Steffen and T. Margaria, editors, *Proceedings of the International Workshop on Software Tools for Technology Transfer STTT '98*, pages 13–24, Aalborg, Denmark, 1998.
21. H. Emmelmann. Code selection by regularly controlled term rewriting. In R. Giegerich and S.L. Graham, editors, *Code Generation - Concepts, Tools, Techniques*, Workshops in Computing. Springer-Verlag, 1992.
22. H. Emmelmann, F.-W. Schröer, and R. Landwehr. Beg - a generator for efficient back ends. In *ACM Proceedings of the Sigplan Conference on Programming Language Design and Implementation*, June 1989.
23. Albert Endres. An analysis of errors and their causes in system programs. *SIGPLAN Notices*, 10(6):327–336, 1975.
24. David A. Espinosa. *Semantic Lego*. PhD thesis, Columbia University, 1995.
25. T.S. Gaul. An Abstract State Machine Specification of the DEC-Alpha Processor Family. Verifix Working Paper [Verifix/UKA/4], Universität Karlsruhe, 1995.
26. W. Goerigk, T.S. Gaul, and W. Zimmermann. Correct Programs without Proof? On Checker-Based Program Verification. In *Proceedings ATOOLS'98 Workshop on "Tool Support for System Specification, Development, and Verification"*, Advances in Computing Science, Malente, 1998. Springer Verlag.
27. Wolfgang Goerigk. Towards Rigorous Compiler Implementation Verification. In *Proc. of the 1997 Workshop on Programming Languages and Fundamentals of Programming*, Avendorf, Germany, November 1997.
28. Gerhard Goos. Sather-K — The Language. *Software — Concepts and Tools*, 18:91–109, 1997.
29. Y. Gurevich. Evolving Algebras: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
30. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In *CSL '92*, volume 702 of *LNCS*, pages 274–308. Springer-Verlag, 1993.
31. A. Heberle, T. Gaul, W. Goerigk, G. Goos, and W. Zimmermann. Construction of Verified Compiler Front-Ends with Program-Checking. In *Proceedings of PSI '99: Andrei Ershov Third International Conference on Perspectives Of System Informatics*, volume 1755 of *Lecture Notes in Computer Science*, Novosibirsk, Russia, 1999. Springer Verlag.
32. Andreas Heberle and Dirk Heuzeroth. The formal specification of IS. Technical Report [Verifix/UKA/2 revised], IPD, Universität Karlsruhe, January 1998.
33. C.A.R. Hoare, He Jifeng, and A. Sampaio. Normal Form Approach to Compiler Design. *Acta Informatica*, 30:701–739, 1993.
34. J. Huggins and D. Van Campenhout. Specification and Verification of Pipelining in the ARM2 RISC Microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, October 1998.
35. P. W. Kutter and A. Pierantonio. Montages specifications of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

36. John McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In Schwartz [51], pages 33–41.
37. Sun Microsystems. Sun official java compiler bug database. http://java.sun.com/products/jdk/1.2/bugs.html, mar 2000.
38. J S. Moore. *Piton, A Mechanically Verified Assembly-Level Language*. Kluwer Academic Publishers, 1996.
39. C. Robert Morgan. *Building an Optimizing Compiler*. Digital Press, Februar 1998. ISBN 155558179X.
40. P. D. Mosses. Abstract semantic algebras. In D. Bjørner, editor, *Formal description of programming concepts II*, pages 63–88. IFIP IC-2 Working Conference, North Holland, 1982.
41. P. D. Mosses. *Action Semantics*. Cambridge University Press, 1992.
42. Markus Müller-Olm. An Exercise in Compiler Verification. Internal report, CS Department, University of Kiel, 1995.
43. Markus Müller-Olm. *Modular Compiler Verification*. PhD thesis, Techn. Fakultät der Christian-Albrechts-Universität, Kiel, June 1996.
44. Jens Palsberg. *Provably Correct Compiler Generation*. PhD thesis, Department of Computer Science, University of Aarhus, 1992. xii+224 pages.
45. L. Paulson. *A compiler generator for semantic grammars*. PhD thesis, Stanford University, 1981.
46. A. Pnueli, M. Siegel, and E. Singermann. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.
47. Amir Pnueli, O. Shtrichman, and M. Siegel. The code validation tool (cvt). *Int. J. on Software Tools for Technology Transfer*, 2(2):192–201, 1998.
48. W. Polak. Compiler specification and verification. In *Lecture Notes in Computer Science*, number 124 in LNCS. Springer-Verlag, 1981.
49. Robert M. Poston. Preventing software requirements specification errors with ieee 830. *IEEE Software*, 2(1):83–86, January 1985.
50. T. Rus. Algebraic processing of programming languages. *Theoretical Computer Science*, 199:105–143, 1998.
51. J. T. Schwartz, editor. *Mathematical Aspects of Computer Science*, Proc. Symp. in Appl. Math., RI, 1967. Am. Math. Soc.
52. Reinier Sterkenburg. Borland pascal compiler bug list. http://www.dataweb.nl/ r.p.sterkenburg/bugsall.htm, feb 2000.
53. M. Tofte. *Compiler Generators*. Springer-Verlag, 1990.
54. William M. Waite and Gerhard Goos. *Compiler Construction*. Springer-Verlag, 1984.
55. C. Wallace. The Semantics of the C++–Programming Language. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.
56. M. Wand. A semantic prototyping system. *SIGPLAN Notices*, 19(6):213–221, June 1984. SIGPLAN 84 Symp. On Compiler Construction.
57. Hal Wasserman and Manuel Blum. Software reliability via run-time result-checking. *Journal of the ACM*, 44(6):826–849, November 1997.
58. W. Zimmermann and T. Gaul. An Abstract State Machine for Java Byte Code. Verifix Working Paper [Verifix/UKA/12], University of Karlsruhe, 1997.

An ASM Dynamic Semantics for Standard ML

Steven C. Cater and James K. Huggins

Kettering University
Computer Science Program
1700 W. Third Avenue
Flint, MI 48504-4898 USA
{scater,jhuggins}@kettering.edu

Abstract. The Abstract State Machines (ASM) methodology is a methodology for formally specifying computing systems. We use the ASM methodology to give the dynamic semantics of the functional programming language Standard ML. We give an operational semantics for Standard ML by means of an interpreter for (appropriately pre-processed) Standard ML programs; the effect of a Standard ML instruction can be seen in terms of the corresponding actions performed by the ASM.

1 Introduction

The Abstract State Machines (ASM) methodology [11] is a methodology for formally specifying computing systems (software, hardware, or mixed). First introduced by Gurevich [9] (under their former name, "evolving algebras") the ASM methodology is mathematically precise, yet general enough to be applicable to a wide variety of problem domains [3,7,15]. The ASM Thesis [12] asserts that any computing system can be described at its natural level of abstraction by an appropriate ASM.

Standard ML (SML) is a functional programming language. It has been described as a "safe, modular, strict, functional, polymorphic programming language with compile-time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules." [18]

In this paper, we describe the dynamic semantics of SML using the ASM methodology, using Milner [19] as our definition of SML. We describe the dynamic semantics of SML by describing an ASM which acts as an SML interpreter, executing an (appropriately pre-processed) SML program. This provides an operational semantics for SML; the effect of a given SML instruction can be seen in terms of the corresponding actions performed by the ASM.

We separate concerns in this work and restrict our attention to the dynamic semantics of SML, assuming that all static semantic analysis and checking has been performed in a pre-processing stage. We also follow Milner [19] in restricting our attention to the so-called "bare" language of SML: a subset of SML from which all other language constructs in SML can be derived. (For example, arbitrary tuples of the form $(t_1, t_2, \ldots, t_n)$ are translated into records of the form

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 203–222, 2000.

$\{1 = t_1, 2 = t_2, \ldots, n = t_n\}$.) Notice that this does not minimize the scope of the work; using appropriate substitutions, any SML program can be translated into the bare language and thus given semantics by our ASM descriptions.

For brevity, we omit discussion of SML constructs dealing with structure restrictions, functor applications, layered patterns, declarations, and value bindings; a full discussion of these constructs may be found in [6].

We assume familiarity with sequential ASMs in the following sections; see [10] for an introduction to sequential ASMs.

2 ASM for SML: Simple Expressions

We now describe an ASM which acts as an intepreter for SML programs. Since we are solely concerned with the dynamic semantics of SML rather than the static semantics, we assume that all static analysis has already been performed on the program to be interpreted. The input program will thus be represented in an abstract form (to be described). The ASM operates by walking through this form of the program, performing the required calculations and changes to the system state.

In this section, we focus on the evaluation of arbitrary SML expressions written in the "bare" SML language. In succeeding sections we expand our focus to include other features of SML.

2.1 Some Common Universes, Functions, and Abbreviations

An SML program is a collection of terms; consequently, we define a universe of *Terms* which is used to represent a given program. The inter-relationship between various terms in a program are described by various unary functions, most of which are described as they are needed. For example, we use the unary function *Next: Terms* $\rightarrow$ *Terms* to indicate the next term in a sequence of terms being evaluated linearly. Similarly, we use the unary function *Parent: Terms* $\rightarrow$ *Terms* to indicate the smallest enclosing term of a given term. For example, (the term representing) the expression "1+2" is the parent of (the term representing) the expression "1".

The nullary function *CurTerm: Term* is used to indicate the current term being evaluated (similar to the role of an instruction counter). *CurTerm* acts as the focus of attention for the ASM; the ASM at each step examines various information regarding *CurTerm* and makes the appropriate changes to the state.

To identify particular terms, we define a universe of *Kinds* and a unary function *Kind: Terms* $\rightarrow$ *Kinds* used to label individual terms. The members of *Kinds* will be specified as we proceed. For example, a term t representing a function application satisfies *Kind(t) = functionApp*, where *functionApp* is a nullary function indicating a unique element of *Kinds*. (A more intuitive name for *Kinds* might be "*Types*"; we choose not to use that name to avoid confusion with SML's extensive type system.)

```
if OK and CurTerm.Kind = constant then
    CurTerm.Value := CurTerm.ConstValue
    KEEPGOING
endif
```

Fig. 1. Evaluating constants.

The result of evaluating an SML expression is a value; consequently, we define a universe of *Values* comprising the set of possible results of expression evaluation. We use a unary function *Value: Terms* → *Values* to record the results of evaluating a particular term during the course of a computation; the ASM updates *Value* throughout the computation as values propagate from child terms to parent terms.

SML uses special values called exceptions to indicate the presence of errors during a computation; the existence of an exception alters the normal flow of control. We use a universe *Exceptions* (a subset of *Values*) to represent the set of these special values; a nullary function *CurException: Exceptions* indicates the current exception (value) which is propagating through the system. If no exception is propagating, *CurException* has the special value *undef*.

We use two simple abbreviations in our rules. *OK* abbreviates the term "*CurException = undef*"; it indicates that the computation in process is proceeding normally (*i.e.*, no exception is present). *KEEPGOING* abbreviates the rule "*CurTerm := CurTerm.Next*"; it is used when the current term has been processed and attention should be passed to the next term in the sequence of terms. These abbreviations are introduced to improve the readability of the forthcoming rules; additionally, we will later re-define the KEEPGOING abbreviation as the need for a more complicated "keep-going" command becomes necessary.

2.2 Constant Expressions

Evaluating a constant requires almost no work; the result of evaluating a constant is the corresponding constant value. We use a static function *ConstValue: Terms* → *Values* to map a constant expression to its corresponding value. The corresponding ASM rule, shown in Fig. 1, is straightforward.

2.3 Identifiers

In SML, all expressions are evaluated with respect to an *environment*: a finite collection of name bindings. Evaluating an identifier in SML results in the corresponding value stored in the current environment. Thus, to give the rules for identifier evaluation, we need to define how environments are represented in our ASM.

We use universes of *Envs* (for environments) and *EnvEntries* (for entries in an environment) to represent a given environment. An environment contains a

```
if OK and CurTerm.Kind = identifier then
    CurTerm.Value := Lookup(CurTerm.Identifier, CurEnv).ValueEntry
    KEEPGOING
endif
```

Fig. 2. Identifiers.

finite number of entries, indexed by identifier names; we use a universe of *Identifiers* to represent these names. The binary function *Lookup: Envs* × *Identifiers* → *EnvEntries* returns the environment entry corresponding to the specified identifier in the specified environment. The nullary function *CurEnv: Envs* indicates the environment currently being used for evaluating expressions.

Environment entries are not simply values; each entry also has an additional datum indicating whether the value corresponds to a value variable, a value constructor, or an exception constructor. We use a universe of *ValueTags* to indicate these kinds of data in an environment, along with unary (projection) functions *ValueEntry: EnvEntries* → *Values* and *TagEntry: EnvEntries* → *ValueTags* to extract the needed information from a given environment entry.

Having described how environments work in SML, the rule for evaluating an identifier simply accesses the identifier (given by a simple function *Identifier: Terms* → *Identifiers*) and extracts the corresponding value from the current environment. The rule is shown in Fig. 2.

2.4 Records

In SML, a record is represented as a sequence of label-expression pairs; the result of evaluating a record is a finite mapping of the labels in the sequence to the corresponding values (as evaluated in the current environment).

Thus, we need a universe of *Maps* corresponding to these finite pairs of labels and expressions. (Note that *Maps* is not the same universe as *Envs*; environments carry additional information which record maps do not.) We use functions *CreateMap: Identifiers* × *Values* → *Maps* and *AddToMap: Maps* × *Identifiers* × *Values* → *Maps* to construct these maps.

Record expressions are represented in our ASM as a labeled expression optionally followed by a record expression (following the natural recursive definition of a sequence). Our ASM rules for evaluating records (shown in Fig. 3) move through this sequence of record expressions, constructing the corresponding map. (The rule below uses functions *Expr: Terms* → *Terms* and *NextRecord: Terms* → *Terms* to indicate the two components of a record expression; we implicitly introduce component functions such as these in future rules without comment.)

2.5 Reference Terms

In SML, references are pointers to storage locations maintained in system memory. References are used to store and retrieve values outside of the usual en-

```
if OK and CurTerm.Kind = recordExpr then
    if CurTerm.Expr.Value = undef then CurTerm := CurTerm.Expr
    elseif CurTerm.NextRecord = undef then
        CurTerm.Value := CreateMap(CurTerm.Label, CurTerm.Expr.Value)
        CurTerm.Expr.Value := undef
        KEEPGOING
    elseif CurTerm.NextRecord.Value = undef then
        CurTerm := CurTerm.NextRecord
    else
        CurTerm.Value := AddToMap(CurTerm.NextRecord.Value,
                                  CurTerm.Label, CurTerm.Expr.Value)
        CurTerm.Expr.Value := undef, CurTerm.NextRecord.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 3. Records.

```
if OK and CurTerm.Kind = reference then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    else
        extend Addresses with a
            CurTerm.Value := a, Store(a) := CurTerm.Argument.Value
        endextend
        CurTerm.Argument.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 4. Reference terms.

vironment and expression evaluation mechanism; they thus break the "pure" functional programming paradigm.

Evaluating reference term of the form "`ref t`" causes a new memory location to be allocated and returned as the value of the expression. Additionally, the new memory location is initialized to the value of the argument expression `t`, as evaluated in the current environment.

Thus, we need a new universe of *Addresses* to represent references, and a unary function *Store: Addresses* → *values* to represent the current memory store. Evaluating a reference term thus requires allocating a new memory address and initializing the store appropriately. The resulting rule is shown in Fig. 4.

```
if OK and CurTerm.Kind = assign then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    else
        Store(MapLookup(CurTerm.Argument.Value,"1")) :=
            MapLookup(CurTerm.Argument.Value,"2"))
        CurTerm.Value := Unit, CurTerm.Argument.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 5. Assignment expressions.

2.6 Assignment

Assignment terms change the values stored in system memory. Syntactically, an assignment appears in the "bare" language as the application of the special function ":=" to a single value: a record in which the label "1" is bound to the memory address to be changed and the label "2" is bound to the new value.

Recall that the value of a record is a map. In order to extract the necessary values from the argument record, we use a binary function *MapLookup: Maps* × *Identifiers* → *Values*. The rule (shown in Fig. 5) extracts the necessary information and performs the assignment.

All expressions in SML evaluate to a value; however, assignment expressions are evaluated for their side-effects rather than any value they might return. SML uses the special value "unit" to represent the value of an assignment expression; we use a corresponding distinguished element *Unit: Values*.

2.7 Raising and Propagating Exceptions

A "`raise`" expression takes an argument which evaluates to an exception value. Evaluating such an expression causes the argument to be evaluated; the resulting value is then set to propagate through the system (in our ASMs, this is performed by assigning the value to *CurException*).

Exceptions propagate by moving from child term to parent term repeatedly until a `handle` expression is found. The rules shown in Fig. 6 show how exceptions are raised and propagated. Note that the presence of a propagating exception in *CurException* falsifies the *OK* term, thus making most other rules inapplicable in this situation.

3 ASM for SML: Function Applications

In this section, we focus on the evaluation of SML expressions involving function application, which involve evaluating expressions in environments other than the current environment. We also discuss other SML expressions whose semantics involve changing the current evaluation environment.

```
if OK and CurTerm.Kind = raise then
    if CurTerm.Argument.Value = undef then CurTerm := CurTerm.Argument
    else
        CurException := CurTerm.Argument.Value
        CurTerm.Argument.Value := undef, CurTerm := CurTerm.Parent
    endif
endif
if CurException ≠ undef and CurTerm.Kind ≠ handle then
    CurTerm.Value := undef, CurTerm := CurTerm.Parent
endif
```

Fig. 6. Raising and propagating exceptions.

```
    if OK and CurTerm.Kind = functionClosure then
        CurTerm.Value := MakeFunction (CurTerm.MatchTerm, CurEnv)
        KEEPGOING
    endif
```

Fig. 7. Function closures.

3.1 Function Closures

A function closure is a match rule (in brief, a pattern-matching rule for evaluating the function) along with the environment to be used in evaluating that match rule. Function closures are created by evaluating statements of the form "`fn` *match*", where *match* is a match rule. The environment currently in use is bound to the specified match rule for later use.

We consequently make use of a universe of *FunctionClosures* representing this information, with construction function *MakeFunction: Terms* × *Envs* → *FunctionClosures* and projection functions *MatchBinding: FunctionClosures* → *Terms* and *EnvBinding: FunctionClosures* → *Envs*. The rule for evaluating function closure expressions is shown in Fig. 7.

3.2 Function Applications: Preliminaries and Special Cases

Several forms of function application are present in SML. In all of these cases, a function application consists of two terms; a term corresponding to the function to be applied, and a term corresponding to the argument for that function. (All functions in the "bare" language are unary; functions of higher arity are simulated by records.) The rule shown in Fig. 8 performs this evaluation in all cases.

One special case occurs when the function expression is not a function at all, but a value constructor (*i.e.*, an identifier). In this case, the value of the expression is a pair consisting of the constructor and the argument value. We

```
if OK and CurTerm.Kind = functionApp then
    if CurTerm.LeftExpr.Value = undef then CurTerm := CurTerm.LeftExpr
    elseif CurTerm.RightExpr.Value = undef then CurTerm := CurTerm.RightExpr
    endif
endif
```

Fig. 8. Evaluating arguments for function application.

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ undef
then
    if CurTerm.LeftExpr.Value.Identifier then
        CurTerm.Value := Pair(CurTerm.LeftExpr.Value, CurTerm.RightExpr.Value)
        CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 9. Constructor application.

use the function *Pair: Values* × *Values* → *Values* to construct such pairs. The rule for constructor application is shown in Fig. 9.

A second special case occurs when the function to be applied is a built-in operator. We use a universe *PrimitiveOps* to represent such primitive operations, as well as a function *Apply: PrimitiveOps* × *Values* → *Values* to represent the definition of such primitive operators. Thus, our rule simply uses the *Apply* function to generate the appropriate value.

Applying a primitive operator may also generate an exception; thus, if the value returned by *Apply* is an exception, we need to initiate exception handling (as in the case of a `raise` expression). Thus, the rule shown in Fig. 10 checks the resulting value before passing it along.

3.3 Function Application to Function Closures

Here we handle the case of applying a user-defined function closure to an argument term. As seen above, function closures include a match expression and a current environment. Evaluating such a function application involves evaluating the specified match expression against the value of the function argument; however, such evaluation occurs not in the current environment, but in the environment specified in the function closure. (We defer our discussion of evaluating match terms against values.)

Of course, evaluating this match term in the new environment could involve evaluating other function applications, requiring evaluating other match terms in other environments, and so on. It becomes clear that we need a stack-like

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ undef
then
    if CurTerm.LeftExpr.Value.PrimitiveOp then
        let Result = Apply(CurTerm.LeftExpr.Value, CurTerm.RightExpr.Value) in
            if Result.Exception then CurException := Result
            else
                CurTerm.Value := Result
                KEEPGOING
            endif
        endlet
        CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
    endif
endif
```

Fig. 10. Primitive operator application.

structure to use in evaluating terms in environments different from the current environment, while still maintaining the current environment for future use.

The elements which need to be stored in an activation record upon our stack are the following: the intermediate expression values already generated and stored in the function *Value*, the current focus of attention stored in *CurTerm*, the current evaluation environment stored in *CurEnv*, and the current exception propagating stored in *CurException*. Rather than creating an explicit stack structure (and following the precedent in [13]), we use the universe *Naturals* of natural numbers and redefine the functions named above to have the following signatures:

Value: Terms × *Naturals* → *Values* *CurTerm: Naturals* → *Terms*
CurException: Naturals → *Exceptions* *CurEnv: Naturals* → *Envs*

The extra numerical argument to each of these functions is used to indicate the level of the stack at which the given value is stored. For example, *CurTerm(3)* indicates the term currently being evaluated at level 3 of the stack. A new distinguished function *StackLevel: Naturals* indicates the current position on the stack; initially, *StackLevel = 0*. Thus, *CurTerm(StackLevel)* indicates the term currently being evaluated.

This requires changes to all the rules previously presented. In order to simplify our presentation (and following the precedent in [4]), we abbreviate *CurTerm(StackLevel)* by *CurTerm*, *Value(X,StackLevel)* by *Value(X)*, and so on. In any function hereafter with an argument belonging to *Naturals*, we suppress that argument if its value is *StackLevel*; we explicitly specify the argument if it is a term other than *StackLevel* or if clarity would be better served by explicit specification. This set of abbreviations allows us to ignore the stack in situations where its presence is semantically uninteresting, and also allows us to use the previous rules without modification (other than, of course, the above abbreviations).

```
EVALUATE(term, env, matchval) ::=
    Target(StackLevel + 1) := term
    CurTerm(StackLevel + 1) := term
    CurEnv(StackLevel + 1) := env
    CurMatchVal(StackLevel + 1) := matchval
    StackLevel := StackLevel + 1

EVALUATE(term, env) ::= EVALUATE(term, env, undef)
```

Fig. 11. *EVALUATE* abbreviation.

```
if OK and CurTerm.Kind = functionApp and CurTerm.RightExpr.Value ≠ undef
      and CurTerm.LeftExpr.Value.FunctionClosure then
   if ReturnValue = undef then
      EVALUATE(CurTerm.LeftExpr.Value.MatchBinding,
               CurTerm.LeftExpr.Value.EnvBinding,
               CurTerm.RightExpr.Value)
   else
      CurTerm.Value := ReturnValue, ReturnValue := undef
      CurTerm.LeftExpr.Value := undef, CurTerm.RightExpr.Value := undef
      KEEPGOING
   endif
endif
```

Fig. 12. Function application to function closures.

We introduce some functions used in the process of making a context-switch dealing with the call stack. *Target: Naturals* → *Terms* indicates the top-level term which is to be evaluated at this level of the stack. *CurMatchValue: Naturals* → *Values* indicates the value which is to be matched against *Target* as part of this evaluation process. *ReturnValue: Values* is used to store the return value when a given term evaluation on the stack completes.

We often make use of the *EVALUATE* abbreviation, shown in Fig. 11. The idea is that *EVALUATE(t, e, v)* attempts to match term t against value v using environment e. The abbreviation works by implicitly creating the next entry on the call stack and transferring control to term t. When that term completes its evaluation, *ReturnValue* will have the desired result and *StackLevel* will be reset to the proper value. The alternate form *EVALUATE(t,e)* is used in situations where a match value is not required.

Finally, we can present the rule for performing a function application to a function closure. The rule shown in Fig. 12 simply invokes the specified function closure, attempting to match the specified argument against the function closure term.

```
KEEPGOING ::=
    if CurTerm = Target then CurTerm := ReturnTerm
    else CurTerm := CurTerm.Next
    endif
```

Fig. 13. New definition of the *KEEPGOING* abbreviation.

```
if CurTerm = ReturnTerm then
    if CurException = undef then ReturnValue := Value(Target)
    else
        ReturnValue := CurException, CurException(StackValue-1) := CurException
    endif
    Target.Value := undef, CurException := undef
    StackLevel := StackLevel - 1
endif
```

Fig. 14. Returning from a function call.

3.4 Returning from Function Closures

The rules presented in the last section provide only half of what is needed for handling calls to function closures. The rules show how to invoke the evaluation of another expression in another environment; they fail to show what to do when evaluation of that expression has been completed. Here we complete the picture.

We begin by re-defining the abbreviation *KEEPGOING*. Recall that KEEPGOING works by moving *CurTerm* to the next expression to be evaluated in the program. This behavior should be suspended when *CurTerm* has returned to *Target*; at this point, the value stored at *Target* should be returned to the caller. The first half of this process is shown in the new definition of *KEEPGOING* in Fig. 13; it transfers control in such situations to a new distinguised element *ReturnTerm: Terms*, where we perform the actual return.

We now show what happens when control reaches *ReturnTerm*. Here we simply have to keep the promises we made earlier: place the proper return value into *ReturnValue* and re-establish the previous evaluation context. Note that we have to deal with situations in which an exception is propagating as well; in such situations, we return the exception in both *ReturnValue* and *CurException*. The rule is shown in Fig. 14.

3.5 Handle Expressions

A handle expression is used to encapsulate exception processing over a given term. The argument term is evaluated; if no exception occurs while evaluating the argument, the handle expression does nothing and control continues on as usual. The rule for this case is shown in Fig. 15; notice that the truth of the guard

```
if OK and CurTerm.Kind = handle then
    if CurTerm.Expr.Value = undef then CurTerm := CurTerm.Expr
    else
        CurTerm.Value := CurTerm.Expr.Value, CurTerm.Expr.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 15. Normal processing of handle expressions.

```
        if CurException ≠ undef and CurTerm.Kind = handle then
            if ReturnValue = undef then
                EVALUATE(CurTerm.Match, CurEnv, CurException)
            elseif ReturnValue = Fail or ReturnValue.Exception then
                CurTerm.Value := undef, CurTerm := CurTerm.Parent
            else
                CurTerm.Value := ReturnValue, CurException := undef
                KEEPGOING
            endif
            ReturnValue := undef
        endif
```

Fig. 16. Exception processing of handle expressions.

OK indicates that no exceptions have occurred while processing the argument term.

Associated with each handle expression is a match rule (similar to that used in function definitions). If a propagating exception reaches a handle expression, we attempt to match the propagating exception (which is, after all, a value) against the associated match rule, as if we made a function call to the match rule. If the match rule returns a value or an exception, we allow that value or exception to continue propagating. The special value *Fail: Values* may be returned if the match rule fails to generate a result; in such situations, we allow the old exception to continue propagating. We use our context-switch mechanism introduced in the last section to perform this evaluation. The corresponding rule is shown in Fig. 16.

3.6 Let Expressions

Let expressions contain a set of bindings (which generate an environment) and an expression which should be evaluated with respect to those bindings and the current environment. Those bindings should only be added to the current environment to evaluate the target expression; afterwards, the current environment should revert its previous value.

```
if OK and CurTerm.Kind = let then
    if CurTerm.LeftExpr.Value = undef then CurTerm := CurTerm.LeftExpr
    elseif ReturnValue = undef then
        EVALUATE(CurTerm.RightExpr,
                 CombineEnv(CurEnv, CurTerm.LeftExpr.Value))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.LeftExpr.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 17. Let expressions.

We can use the same context-switch mechanism introduced previously to provide semantics for let expressions. We first evaluate the set of attached bindings, which generates an environment. We then combine that environment with the current environment, using a function *CombineEnv: Envs* × *Envs* → *Envs* to perform the combination appropriately. The new combined environment is then used as the basis for a context-switch to evaluate the target expression; thus, the new bindings implicitly disappear (as desired) when the call returns. The rule for evaluating let expressions is shown in Fig. 17.

4 ASM for SML: Matches

In this section, we focus on the evaluation of SML match rules. An SML match rule consists of a pattern and an associated expression. Match rules are always evaluated with respect to a target value. A successful attempt to match a value against a pattern results in an environment, representing the bindings required to match the value against that pattern; in such situations, the associated expression is then evaluated with respect to the current environment augmented by the new bindings, and its value returned. An unsuccessful attempt to match a value against a pattern results in the special value *Fail: Values*.

4.1 Match Lists

Matches usually occur in a list. Evaluating a list of matches is relatively straightforward; one evaluates each match in the list sequentially until a non-*Fail* value is generated, which is returned as the value of the list expression. Should all matches in the list return *Fail*, the list returns *Fail* as well.

The rule for evaluating match lists is shown in Fig. 18. Note that we associate with every match list its component rule (found with *MatchRule: Terms* → *Terms*) and the remainder of the list which follows that rule (found with *MatchList: Terms* → *Terms*).

```
if OK and CurTerm.Kind = matchList then
    if CurTerm.MatchRule.Value = undef then CurTerm := CurTerm.MatchRule
    elseif CurTerm.MatchRule.Value ≠ Fail then
        CurTerm.Value := CurTerm.MatchRule.Value
        CurTerm.MatchRule.Value := undef
        KEEPGOING
    elseif CurTerm.MatchList = undef then
        CurTerm.Value := Fail, CurTerm.MatchRule.Value := undef
        KEEPGOING
    elseif CurTerm.MatchList.Value = undef then CurTerm := CurTerm.MatchList
    else
        CurTerm.Value := CurTerm.MatchList.Value
        CurTerm.MatchList.Value := undef, CurTerm.MatchRule.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 18. Match lists.

4.2 Match Rules

The semantics for evaluating match rules were explained in the introduction to this section; the corresponding rule is shown in Fig. 19.

4.3 Default Patterns

Here we begin to present rules for handling various types of patterns. Recall that *CurMatchVal* is used to hold the value against which the current pattern is to be matched; the result of a pattern match is either an environment or the special value *Fail*.

The simplest patterns to match are the underscore pattern "_" and the ellipsis pattern "...". Either pattern matches against any value and creates no bindings; the empty environment is returned. We use the distinguished element *EmptyEnv: Envs* to represent the environment containing no bindings. The corresponding simple rule is shown in Fig. 20.

4.4 Constant Patterns

One can match values against special patterns representing constants (*e.g.*, numeric literals). The match succeeds if and only if the constant pattern corresponds to the current match value; no bindings are generated. Using the function *ConstValue: Terms* → *Values* to indicate the true value of this constant term, the rule shown in Fig. 21 performs this operation.

```
if OK and CurTerm.Kind = matchRule then
    if CurTerm.Pattern.Value = undef then CurTerm := CurTerm.Pattern
    elseif CurTerm.Pattern.Value = Fail then
        CurTerm.Value := Fail, CurTerm.Pattern.Value := undef
        KEEPGOING
    elseif ReturnValue = undef then
        EVALUATE(CurTerm.Expr, CombineEnv(CurEnv, CurTerm.Pattern.Value))
    else
        CurTerm.Value := ReturnValue, ReturnValue := undef
        CurTerm.Pattern.Value := undef
        KEEPGOING
    endif
endif
```

Fig. 19. Match rules.

```
        if OK and (CurTerm.Kind = "_" or CurTerm.Kind = "...") then
            CurTerm.Value := EmptyEnv
            KEEPGOING
        endif
```

Fig. 20. Default patterns.

4.5 Simple Identifiers

Matching a value against an identifier in the current environment succeeds in three cases:

1. The identifier is currently unbound. The resulting value is a single-entry environment, binding the identifier to the value.
2. The identifier is currently bound to a free variable. The resulting value is a single-entry environment, binding the identifier to the variable.
3. The identifier is already bound to the desired value. No additional bindings are generated.

```
  if OK and CurTerm.Kind = specialConstant then
      if CurMatchVal = CurTerm.ConstValue then CurTerm.Value := EmptyEnv
      else CurTerm.Value := Fail
      endif
      KEEPGOING
  endif
```

Fig. 21. Constant patterns.

```
if OK and CurTerm.Kind = identifier then
    let id = CurTerm.Identifier in
        if Lookup(CurEnv, id) = undef
        or Lookup(CurEnv, id).TagEntry = ValueVariable then
            CurTerm.Value := CreateEnv(id, Pair(CurMatchVal, ValueVariable))
        elseif Lookup(CurEnv, id).ValueEntry = CurMatchVal then
            CurTerm.Value := EmptyEnv
        else CurTerm.Value := Fail
        endif
        KEEPGOING
    endlet
endif
```

Fig. 22. Identifier patterns.

We use a new function *CreateEnv: Identifiers* $\times$ *Values* $\rightarrow$ *Envs* to create a single-entry environment with the specified binding. The corresponding rule is shown in Fig. 22.

4.6 Records: Labeled Patterns

Recall that the value of a record in SML is a finite name/value mapping. To match a record value against a sequence of labeled patterns, we simply need to ensure that the labeled sequence agrees with the record. That is, if label ℓ is associated with expression e, the current match value should associate ℓ with a value v such that v successfully (recursively) matches against e.

Consequently, our context-switch rules come in handy again, as we need to repeatedly check that each labeled expression successfully matches against the corresponding values in the match record. Our rule, shown in Fig. 23, simply proceeds along the list of labeled patterns, combining the resulting environments as needed.

4.7 Constructed Patterns

Constructed patterns are patterns consisting of a constructor and an argument. A constructed pattern successfully matches against a value which is itself a constructed value with the same identifier and whose value (recursively) matches against the pattern argument. The rule for constructed patterns is shown in Fig. 24.

4.8 Memory References

A memory reference term successfully matches against a value if the value is an address, and the value stored at that address in memory (recursively) matches against the argument of the reference term. The rule for memory reference terms is given in Fig. 25.

```
if OK and CurTerm.Kind = labeledPattern then
    if ReturnValue = undef then
        EVALUATE(CurTerm.Pattern, CurEnv,
                 MapLookup(CurMatchVal, CurTerm.Label))
    else
        if ReturnValue = Fail then
            CurTerm.Value := Fail
            KEEPGOING
        elseif CurTerm.MoreBindings = undef then
            CurTerm.Value := ReturnValue
            KEEPGOING
        elseif CurTerm.MoreBindings.Value = undef then
            CurTerm.Pattern.Value := ReturnValue
            CurTerm := CurTerm.MoreBindings
        else
            if CurTerm.MoreBindings.Value = Fail then CurTerm.Value := Fail
            else CurTerm.Value := CombineEnv(CurTerm.Pattern.Value,
                                             CurTerm.MoreBindings.Value)
            endif
            CurTerm.Pattern.Value := undef
            CurTerm.MoreBindings.Value := undef
            KEEPGOING
        endif
    endif
endif
```

Fig. 23. Record matching.

5 Discussion

ASMs were first proposed as a methodology for specifying the semantics of programming languages [8]. ASMs have been applied to a wide variety of programming languages: imperative languages such as C/C++ [13,20], logic programming languages such as Prolog [4] and its variants, object-oriented languages such as Java [5,21] and Oberon [17], and hardware languages such as VHDL [2]. To the best of our knowledge, this case study in Standard ML is the first application of ASMs to provide the semantics of a functional programming language.

The official semantics of Standard ML is given by Milner [19], using an axiomatic semantics called Natural Semantics. The rules given in Milner, while definitive, rely heavily on axiomatic notation and proof rules which can be difficult to read. With the presence of an official semantics, there appear to be few other treatments of SML using other semantic techniques. A recent paper by Watt [22] announces the use of Action Semantics to give semantics to SML; another work by Harper and Stone [14] translates SML into a typed variant of the lambda calculus to which an operational semantics is given. All of these other works give both static and dynamic semantics for SML.

```
if OK and CurTerm.Kind = constructedPattern then
    let id = CurTerm.Identifier in
        if Lookup(id, CurEnv).ValueEntry.Second ≠ ValueConstructor
        and Lookup(id, CurEnv).ValueEntry.Second ≠ ExceptionConstructor then
            CurTerm.Value := Fail
            KEEPGOING
        elseif Lookup(id, CurEnv).ValueEntry.First ≠ CurMatchVal.First then
            CurTerm.Value := Fail
            KEEPGOING
        elseif ReturnValue = undef then
            EVALUATE(CurTerm.Argument, CurEnv, CurMatchVal.Second)
        else
            CurTerm.Value := ReturnValue, ReturnValue := undef
            KEEPGOING
        endif
    endlet
endif
```

Fig. 24. Constructed pattern matches.

```
    if OK and CurTerm.Kind = memoryReference then
        if ReturnValue = undef then
            EVALUATE(CurTerm.Argument, CurEnv, Store(CurMatchVal))
        else
            CurTerm.Value := ReturnValue, ReturnValue := undef
            KEEPGOING
        endif
    endif
```

Fig. 25. Reference matches.

One interesting feature of our ASM semantics for SML is our use of the EVALUATE macro, used to evaluate a term in an environment differing from the current environment. The EVALUATE macro appears in roughly one-third of the rules given above (not counting rules omitted for brevity), suggesting that term-environment evaluation plays an important role in the dynamic semantics of SML. This is natural; SML is, after all, a functional programming language, which draws its roots from the lambda calculus. Since function application (*i.e.*, term evaluation in a different environment) is at the heart of the lambda calculus, it seems natural that this feature should exhibit itself so prominently in our ASM semantics.

We would argue that the ASM dynamic semantics given above are as precise and accurate as conventional semantic treatments such as the official definition [19]. The rules and explanations above are somewhat longer than in an axiomatic

approach, as the ASM notation lends itself to natural language notations rather than axiomatic proof rules. However, one may find the readability of these rules is substantially better. Such explanations can provide a better basis for explaining and understanding the language with a minimal amount of notational overhead.

Several possible extensions of this work are being contemplated. One clear gap in the descriptions above is any treatment of the static semantics of SML, including the extensive type system. The official definition of SML [19] spends extensive time defining the static aspects of the language, which can also be confusing at times. Recently Montages [16] have been used to describe both static and dynamic semantics of programming languages using ASMs; we consider extending this work using Montages to provide static semantics for SML as well.

Extending this work to the Montages framework has another important benefit: executability. At this time, the ASM description given here has not been tested by any executable tools. The Gem-Mex tool [1] allows one to execute Montage descriptions directly; we expect that doing so will give us further evidence of the correctness of these descriptions.

Alternatively, ASMs have been used for various proofs of correctness, such as compilation techniques; we consider specifying a compilation technique for SML to some intermediate language and proving its correctness.

References

1. M. Anlauff, P. Kutter, and A. Pierantonio. Formal Aspects of and Development Environments for Montages. In M. Sellink, editor, *2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing, Amsterdam, 1997. Springer.
2. E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE CS Press.
3. E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. (An updated version is available from [15].).
4. E. Börger and D. Rosenzweig. A Mathematical Definition of Full Prolog. In *Science of Computer Programming*, volume 24, pages 249–286. North-Holland, 1994.
5. E. Börger and W. Schulte. Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
6. S. Cater and J. Huggins. An ASM Dynamic Semantics for Standard ML. Technical Report CPSC-1999-2, Kettering University, 1999.
7. U. Glässer. Abstract State Machines Europe Home Page. http://www.uni-paderborn.de/cs/asm/.
8. Y. Gurevich. Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs. Technical Report CRL-TR-38-84, EECS Department, University of Michigan, 1984.
9. Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

10. Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264–284, 1991. (Reprinted in G. Rozenberg and A. Salomaa, eds., *Current Trends in Theoretical Computer Science*, World Scientific, 1993, 266–292.).
11. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
12. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, page to appear, 2000.
13. Y. Gurevich and J. Huggins. The Semantics of the C Programming Language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *LNCS*, pages 274–309. Springer, 1993.
14. Robert Harper and Chris Stone. A type-theoretic interpretation of Standard ML. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Robin Milner Festschrifft*. MIT Press, 1998.
15. J. Huggins. Abstract State Machines Home Page. http://www.eecs.umich.edu/gasm/.
16. P.W. Kutter and A. Pierantonio. Montages: Specifications of Realistic Programming Languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.
17. P.W. Kutter and A. Pierantonio. The Formal Specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
18. Lucent Technology. Standard ML of New Jersey Home Page. http://cm.bell-labs.com/cm/cs/what/smlnj/.
19. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
20. C. Wallace. The Semantics of the C++ Programming Language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.
21. C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.
22. D. Watt. The Static and Dynamic Semantics of Standard ML. In *Proceedings of the Second International Workshop on Action Semantics (AS'99)*, number NS-99-3 in BRICS Notes Series, pages 155–172. Department of Computer Science, University of Aarhus, May 1999.

Modeling the Dynamics of UML State Machines

Egon Börger[1], Alessandra Cavarra[2], and Elvinia Riccobene[2]

[1] Dipartimento di Informatica - Università di Pisa - C.so Italia, 40 - 50125 Pisa
boerger@di.unipi.it
(currently visiting Microsoft Research, Redmond)
[2] Dipartimento di Matematica e Informatica - Università di Catania -
V.le A. Doria, 6 - 95125 Catania
{cavarra, riccobene}@dmi.unict.it

Abstract. We define the dynamic semantics of UML State Machines which integrate statecharts with the UML object model. The use of ASMs allows us (a) to rigorously model the event driven *run to completion* scheme, including the sequential execution of entry/exit actions (along the structure of state nesting) and the concurrent execution of internal activities; (b) to formalize the object interaction, by combining control and data flow features in a seamless way; and (c) to provide a precise but nevertheless provably most general computational meaning to the UML terms of atomic and durative actions/activities. We borrow some features from the rigorous description of UML Activity Diagrams by ASMs in [7].

1 Introduction

The Unified Modeling Language [2,5,20] is a standardized notation based on a set of diagrams to describe the structure and the behavior of a software system. In [5] it is stated that "UML is more than just a graphical language. Rather, behind every part of its graphical notation there is a specification that provides a textual statement of the syntax and *semantics* of that building block" although the official document [4] for the UML semantics only gives an unambiguous textual definition of the syntax for UML notations and leaves the behavioral content of various UML constructs largely open. The necessity to develop the UML as a precise (i.e. well defined) modeling language is widely felt [10,9,19] and the *pUML* (*precise UML*) group has been created to achieve this goal [18].

In this paper we analyze one of the principal diagram types which are used in UML for the description of dynamical system behavior, namely statechart or state diagrams, and provide a rigorous definition of their dynamics. Many papers on the semantics of statecharts [16,21,11,17] exist in the literature, in particular in relation to their implementation in STATEMATE [15] and RHAPSODY [14]. Nevertheless, the debate is still ongoing on what exactly should be considered as the authoritative definition of UML State Machines which integrate statecharts with the UML object model. One major difficulty here concerns the mechanisms for object interaction [14,19].

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 223–241, 2000.

ASMs [1,12] provide a technique to solve such specification problems and to clarify the relevant issues. In this paper, we propose an ASM model that (a) rigorously defines the UML event handling scheme in a way which makes all its "semantic variation points" explicit, including the event deferring and the event completion mechanism; (b) encapsulates the run to completion step in *two* simple rules (**Transition Selection** and **Generate Completion Events**) where the peculiarities relative to entry/exit or transition actions and sequential, concurrent or history states are dealt with in a modular way; (c) integrates smoothly the state machine control structure with the data flow; (d) clarifies various difficulties concerning the scheduling scheme for internal ongoing (really concurrent) activities; (e) describes all the UML state machine features that break the thread-of-control; (f) provides a precise computational content to the UML terms of atomic and durative actions/activities, without loosing the intended generality of these concepts (see footnote 10), and allows one to clarify some dark but semantically relevant points in the UML documents on state machines.

We do not take any position on which UML concepts or understandings of them are reasonable or desirable. Through our definitions we however build a framework for rigorous description and analysis of logically consistent interpretations of the intuitions which underly UML concepts. In fact, exploiting the abstract nature of ASMs it is easy to adapt our definitions to changing requirements. We hope that this will contribute to their rational reconstruction, for the standardization, and to the comparison of different implementations. Our model can also serve as reference model for implementing tools for code generation, simulation and verification of UML models. This work can be viewed as a continuation of [7] where a rigorous semantics of UML activity diagrams has been provided.

The paper is organized as follows. Section 2 introduces the basic concepts underlying UML statechart diagrams and ASMs. The ASM model for the behavioral meaning of these diagrams is defined in section 3. In section 4, the semantical equivalence between some state machine building blocks is discussed. In section 5 we compare our model with related work and show that it satisfies the UML meta-model requirements for state machines.

2 Basic Concepts

In this section we sketch the basic concepts underlying UML state machines and ASMs and review the notation.

2.1 UML Statechart Diagrams

Statechart diagrams are one of the five diagrams in the UML for modeling the dynamic aspects of systems. Statecharts were invented by David Harel [15,16], the semantics and the notation of UML statecharts are substantially those of Harel's statecharts with adaptations to the object-oriented context [3].

Statechart diagrams focus on the event-ordered behavior of an object, a feature which is specially useful in modeling reactive systems. A statechart diagram shows the event triggered flow of control due to transitions which lead from state to state, i.e. it describes the possible sequences of states and actions through which a model element can go during its lifetime as a result of reacting to discrete events. A state reflects a situation in the life of an object during which this object satisfies some condition, performs some action, or waits for some event. According to the UML meta-model [4], states can belong to one of the following categories: *simple states*, *composite states* (sequential, concurrent, submachine), *final*, and *pseudostates* (initial, history, stub, junction, synch).

Transitions are viewed in UML as relationships between two states indicating that an object in the first state will enter the second state and perform specific actions when a specified event occurs provided that certain conditions are satisfied [3]. UML statecharts include *internal*, *external* and *completion* transitions.

The semantics of event processing in UML state machines is based on the *run to completion* (rtc) assumption: events are processed one at a time and when the machine is in a stable configuration, i.e. a new event is processed only when all the consequences of the previous event have been exhausted. Therefore, an event is never processed when the state machine is in some intermediate, unstable situation.

Events may be specified by a state as being possibly deferred. They are actually deferred if, when occurring, they do not trigger any transition. This will last until a state is reached where they are no more deferred or where they trigger a transition.

2.2 Abstract State Machines

ASMs are transition systems, their states are multi-sorted first-order structures, i.e. sets with relations and functions, where for technical convenience relations are considered as characteristic boolean-valued functions. The transition relation is specified by rules describing the modification of the functions from one state to the next, namely in the form of guarded updates ("rules")

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

where *Updates* is a set of function updates $f(t_1, \ldots, t_n) := t$, which are simultaneously executed when *Condition* is true.

We use *multi-agent ASMs* [12,1] to model the concurrent substates and the internal activities which may appear in a UML statechart diagram. A multi-agent ASM is given by a set of (sequential) agents, each executing a program consisting of ASM rules. Their distributed runs are defined in [12].

Since ASMs offer the most general notion of state, namely structures of arbitrary data and operations which can be tailored to any desired level of abstraction, this allows us on the one side to reflect in a simple and coherent way the integration of control and data structures, resulting from mapping statecharts to the UML object model. In fact, machine transitions are described by ASM rules where the actions become updates of data (function values for given arguments).

On the other side also the interaction between objects is naturally reflected by the notion of state of multi-agent (distributed) ASMs.

For the constructs of sequentialization, iteration and submachine of sequential ASMs we use the definitions which have been given in [8]. They provide the concept of "stable" state needed to guarantee that the event triggered sequential exit from and entry into nested diagrams is not interrupted by a too early occurrence of a next event.

3 ASM Model for UML Statechart Diagrams

In this section we model the event governed *run to completion step* in statechart diagrams. We first introduce the signature of UML statecharts and then define the execution rules. Statechart diagrams are made up of (control) states[1] and transitions, belonging to the abstract sets *STATE* and *TRANSITION*.

3.1 Control State

The set *STATE* is partitioned into simple, composite (with substructure), and pseudo states. Composite states are partitioned into sequential and concurrent ones. Pseudostates are partitioned into initial and history states.

Simple states (1) are of the form *state*(*entry,exit,do*(*A*),*defer*)[2], where the parameters *entry*/*exit* denote actions that have to be performed as soon as the state is entered/exited, *do*(*A*) denotes the internal ongoing activity *A* that must be executed as long as the state is active, and *defer* is a set of events that are candidate to be retained in that state.

Sequential composite states (2) are of the form *state*(*entry,exit,do*(*A*),*defer,init,final,history*), where *entry*/*exit*, *do*(*A*) and *defer* have the same meaning as for simple states, *init* denotes the initial state of the submachine state associated to this composite state, *final* denotes its final state, and *history* its associated history state (see below for details on initial, final and history states). Sequential composite states contain one or more substates, exactly one of which is required to be active when the composite state is active.

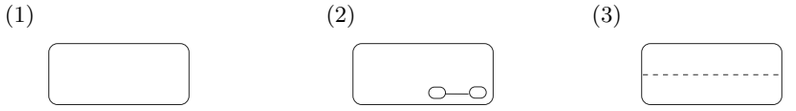
Concurrent composite states (3) are of the form *state*(*entry,exit,do*(*A*),*defer, concurrentComp*), where *entry*/*exit*, *do*(*A*) and *defer* are as above, and *concurrentComp* yields the set of the concurrent (sequential) substates[3] composing the state. When a concurrent state is active, all of its subcomponents are active.

[1] This notion of control state, deriving from the finite state machine notion of "internal" state, is only a tiny fraction of the overall system state which is reflected by the ASM notion of state as structure, i.e. domains (of to be instantiated objects) with operations and relations.

[2] Simple and composite states may have a *name*, i.e. a string denoting uniquely the state. We omit the name parameter in the signature of such states as it is not relevant for our model.

[3] Each substate is sequential because it must enclose an initial and a final state.

According to [4], an event that is deferred in a composite state is automatically deferred in all its directly or transitively nested substates. For reasons of simplicity, but without loss of generality, we assume that the defer set of each state explicitly contains all the inherited events to be deferred.



Initial states • indicate where to start by default when the enclosing (composite sequential) state is invoked. A **History state**, associated to a sequential composite state say S, is a pseudostate that can be of two types: *shallow history* (H) and *deep history* (H*). The shallow history state records, upon exiting S, only the most recent active state directly contained in S and restores the recorded state when the history state is invoked. The deep history state records the most recent active hierarchical configuration of S, and restores this configuration when the history state is invoked. To keep track of the configuration, we use a dynamic function

$$memory : STATE \longrightarrow STATE^*$$

that is initialized to the empty sequence for each state which has never been accessed. To guarantee the correct entering order, we handle *memory* as a LIFO list. In case of shallow history state *memory* contains at most one state.
Final states ⊙ are special states whose activation indicates that the enclosing state is complete.

We denote by *SimpleState, SequentialState, ConcurrentState, PseudoState, FinalState* the characteristic functions of the corresponding subsets of *STATE*.

Any state which is enclosed within a composite state is called a *substate* of that state. In particular, it is called *direct substate* when it is not contained in any other state; otherwise it is referred to as a *transitively nested substate*. The nesting structure of statechart diagrams is encoded by the following static functions:

- *UpState*: $STATE \longrightarrow STATE \cup \{undef\}$, such that $UpState(s) = t$ iff s is a direct substate of a compound state t.
- *DownState*: $STATE \times STATE \longrightarrow BOOL$, such that $DownState(t, s) = true$ iff s is direct substate of a compound state t.
- $UpChain : STATE \times STATE \longrightarrow STATE^*$,

  $UpChain(s, t) = [S_1, \ldots, S_n]$ where $n > 1$ &
  $S_1 = s$ & $S_n = t$ & $\forall i = 2 \ldots n,\ UpState(S_{i-1}) = S_i$

- $DownChain : STATE \times STATE \longrightarrow STATE^*$,

  $DownChain(s, t) = [S_1, \ldots, S_n]$ where $n > 1$ &
  $S_1 = s$ & $S_n = t$ & $\forall i = 1 \ldots n-1,\ DownState(S_i, S_{i+1})$

Upchain and *DownChain* yield empty sequences on each pair of not nested states. We write $Up/DownChain(s_1, \widehat{s_2})$ to indicate the right open sequence $Up/DownChain(s_1, \widehat{s_2}) = [T_1, \ldots, T_n[$, if it exists. Notice that $Up/DownChain(s, \widehat{s}) = []$.

3.2 Transitions

The set *TRANSITION* is partitioned into internal and external transitions.
External transitions are of form *trans*(*source,target,event,guard,action*), where *source*/*target* represent the source/target states of the transition, *event* denotes the triggering event which may enable the transition to fire, *guard* is a boolean expression that is evaluated as soon as the event occurs (if it evaluates to false the transition does not fire), *action* is an action that is executed at the time the transition fires.
Internal transitions are of the form *trans*(*source,event,guard,action*), where all the parameters have the same meaning as for the external transitions. Internal transitions have a source state but no target state because the active state does not change when they fire, and no exit or entry actions are executed. We distinguish between external and internal transitions using a predicate *internal* on *TRANSITION*.
Statechart diagrams include also **completion transitions**, namely transitions with an implicit "completion event" indicating the completion of the state the transition leaves. We can handle completion transitions as special trigger transitions, labeled by *completionEvent*(S), where S is the source state, and assume that all transitions in a statechart diagram are labeled with an event. The only transitions outgoing pseudostates are completion transitions [20].

For each type of state and transition parameter, we use a (static) function *param* which applied to the related states or transitions yields the corresponding parameter. For example *entry*(*state*) yields the entry action associated to *state*, *source*(*trans*) the source state of the transition *trans*, etc. We often suppress parameters notationally.

3.3 Agents

Let *AGENT* be the set of agents which move through the statechart diagram, each executing what is required for its *currently active state*. A state becomes active when it is entered as result of some transition, and becomes inactive if it is exited as result of a transition. "When dealing with composite and concurrent states, the simple term current state can be quite confusing. In a hierarchical state machine more then one state can be active at once. If the control is on a simple state that is contained in a composite state, then all the composite states that either directly or transitively contain the simple state are also active" [4]. Therefore, to maintain what in UML is called the current configuration of active states, we introduce a dynamic function

$$currState : AGENT \rightarrow \mathcal{P}(STATE)$$

whose updates follow the control flow of the given statechart diagram. The function $deepest : AGENT \longrightarrow STATE$ yields the last (innermost) state reached by an agent.

The agents execute UML statechart diagrams, i.e. they all use the same program (or ASM *Rule*). As a consequence, in the formulation of these rules below, we use the 0-ary function *Self* which is interpreted by each agent a as a. When a new agent is created to perform a concurrent subcomputation (defined by one of the substates in a concurrent composite state), it is linked to the *parent* agent by the dynamic function

$$parent : AGENT \rightarrow AGENT \cup \{undef\}$$

We assume that this function yields *undef* for the main agent who is not part of any concurrent flow. The active subagents of an agent a are collected in the set $SubAgent(a) = \{a' \in AGENT \mid parent(a') = a\}$

At the beginning of the computation, we require that there is a unique agent, positioned on the initial state of the *top state*, and whose program consists of the rules **Transition Selection** and **Generate Completion Event** described below.

3.4 Event Handling

In UML it is assumed that a state machine processes one event at a time and finishes all the consequences of that event before processing another event [5, 20]. "An event is *received* when it is placed on the event queue of its target. An event is *dispatched* when it is dequeued from the event queue and delivered to the state machine for processing. At this point, it is referred as the *current event*. Finally, it is *consumed* when event processing is complete. A consumed event is no longer available for processing" [4].

We therefore assume that one event is processed at a time. Since the particular event enqueuing and dispatching mechanisms are deliberately not furthermore specified in UML, we model them here explicitly as semantic variation points and therefore use a monitored predicate *dispatched* indicating which event is dequeued to be processed. At any moment, the only transitions that are eligible to fire when an event e occurs are the ones departing from an active state (i.e. whose source state belongs to *currState*) whose associated guard evaluates to true[4]. This is expressed by the following condition

$$enabled(t, e) \equiv event(t) = e \ \& \ guard(t) \ \& \ source(t) \in currState$$

It is possible for more than one transition to be enabled by the same event, but UML allows only those transitions to be fired simultaneously which occur in concurrent substates [4]. In all the other cases, the enabled transitions are said to be in *conflict* with each other. One can distinguish three types of conflict situations: (1) an internal transition in an active state conflicts with a transition outgoing from that state, (2) two or more transitions originating from the same source in an active state are enabled by e, and (3) two or more transitions with different source states but belonging to the same active state are enabled by the

[4] If no guard is associated to a transition t, we assume $guard(t) = true$.

occurrence of e. In UML the selection among conflicting transitions is constrained only for case (3) by giving priority to the innermost enabled transition. We now formalize this priority for (3), whereas in the cases (1) and (2) we reflect the choice between different scheduling mechanisms as a semantic variation point, namely by using abstract selection functions; see the **Transition Selection** rule below.

Let $enabled(e) = \{t \in TRANSITION \mid enabled(t, e)\}$ be the set of all transitions enabled by e. We define an equivalence relation $\sim$ on $enabled(e)$ as follows: $\forall\ t_1,\ t_2\ \in enabled(e),\ t_1 \sim t_2$ iff $source(t_1) = source(t_2)$.
The nesting of states induces the total order relation[5] $\leq$ on the quotient set $enabled(e)/\sim$, defined as $[t_1] \leq [t_2]$ iff $source(t_1)$ is a direct or a transitively nested substate of $source(t_2)$.
Let $FirableTrans(e)$ be the minimum equivalence class in $enabled(e)/\sim$. It reflects the UML requirement that among transitions enabled by the same event and with different source states, priority is given to an innermost one. The choice among those innermost ones is left open as semantic variation point (see the **choose** construct in the Transition Selection rule).

If a dispatched event does not trigger any transition in the current state, it is lost unless it occurs in the deferred set of the deepest active state. This is formalized by the following predicate *deferrable* on *EVENT*:

$$deferrable(e) = true \Leftrightarrow enabled(e) = \emptyset\ \&\ e \in defer(deepest)$$

As suggested in [20], to store deferred events we associate to each agent a list[6] of events *deferQueue* that is dynamically updated during the computation (see rule Transition Selection). We can therefore define $deferred(e)$ to mean $e \in deferQueue$.

We call a deferred event *releasable* when it becomes ready to be consumed, i.e. when it can trigger a transition in the current state configuration

$$releasable(e) = true \Leftrightarrow deferred(e)\ \&\ enabled(e) \neq \emptyset$$

3.5 Statechart Diagram Main Rules

In this subsection we define the ASM rules for the execution of statecharts, i.e. we specify the sequences of states that an object goes through, and of the actions it takes, in response to events which occur during its lifetime [20].

Apparently, UML leaves it unspecified how to choose between dispatched and releasable events. We reflect this by using a selection function which, at any moment, chooses either a dispatched event triggering a transition, or an event that has been deferred. A dispatched event, if *deferrable*, has to be inserted into the *deferQueue*. A releasable event, when chosen for execution, has to be

[5] Observe that $\leq$ is total since all the source states of the transitions in *enabled* belong to *currState* and therefore they are nested.

[6] Apparently, this list is meant to be a set, leaving the exact ordering of elements open as a semantic variation point. A similar remark applies also to other lists occurring in the UML texts.

deleted from *deferQueue*[7]. This implies that when choosing an event which is simultaneously *dispatched* and *releasable*, that event will be deleted from the deferred events.[8]

We define in the next section the exact meaning of the state machine execution of a transition, namely by a parameterized macro *stateMachineExecution*. This leads us to the following main rule for selecting the machine transition to be executed next.

Rule **Transition Selection**
choose e : *dispatched*(e) $\vee$ *releasable*(e)
  choose *trans* **in** *FirableTrans*(e)
    stateMachineExecution(*trans*)
  if *deferrable*(e) **then** *insert*(e,*deferQueue*)
  if *releasable*(e) **then** *delete*(e,*deferQueue*)

The rule for selecting and executing a transition fires simultaneously, at each "run to completion step", with a rule to generate completion events.

Completion events are generated when an active state satisfies the *completion condition* [4]. They trigger a transition outgoing such states. An active state is considered completed if one of the following cases occurs: (1) it is an active pseudostate, (2) it is a sequential composite state with active final state, (3) the state internal activity terminates while the state is still active, or (4) it is a concurrent composite state and all its direct substates have reached their final state. We formalize this by the predicate

$$\begin{aligned} \mathit{completed}(S) = \mathit{true} \iff\ & \mathit{PseudoState}(S)\ \mathit{or} \\ & (\mathit{SequentialState}(S)\ \&\ \mathit{final}(S) \in \mathit{currState})\ \mathit{or} \\ & \mathit{terminated}(A(S))\ \mathit{or} \\ & (\mathit{ConcurrentState}(S)\ \& \\ & \quad \forall S_i \in \mathit{concurrentComp}(S)\ \forall a_i \in \mathit{SubAgent}(\mathit{Self}) \\ & \qquad \mathit{final}(S_i) \in \mathit{currState}(a_i)) \end{aligned}$$

where *terminated*($A(S)$) is a derived predicate that holds if and only if the *run* of the ASM $A(S)$, which formalizes the internal activity of S, reaches a final state.

Each time the completion condition evaluates to true for an active state S that is not a direct substate of a concurrent state[9] a completion event is generated. This is expressed by the rule **Generate Completion Event** that is executed simultaneously for each state $S \in$ *currState*.

[7] If upon execution of transition *trans*, a deferred event $e \in$ *defer*(*source*(*trans*)) does not belong to *defer*(*target*(*trans*)), then it must be deleted from *deferQueue*, as specified as part of the *enterState* macro below.

[8] Should another interpretation be intended, we would probably change the guard *"if releasable*(e)*"* in the Transition selection rule to e.g. *"if releasable*(e) & *not dispatched*(e)*"*.

[9] This restriction reflects that in UML no direct substate of a concurrent state can generate a transition event. Such substates are required to be sequential composite states.

Rule **Generate Completion Event**
do forall $S \in$ *currState*
if *completed*(S) & $\neg$ *ConcurrentState*(*UpState*(S))
then *generate*(*completionEvent*(S))

Although the order of event dequeuing is not defined, it is explicitly required that completion events must be dispatched before any other queued events [4]. We reflect this requirement as a constraint on the monitored predicate *dispatched*. The above two rules, which fire simultaneously at each *run to completion step*, define the top level behavior of UML state machines. It remains to define in more detail the meaning of the macros appearing in those rules.

The UML requirement that an object is not allowed to remain in a pseudostate, but has to immediately move to a normal state [20], cannot be guaranteed by the rules themselves, but has to be imposed as an integrity constraint on the permissible runs.

3.6 The Rule Macros

We define now the subrule *stateMachineExecution* where parameterization by transitions allows us to modularize the definition for the different types of transitions and the involved states.

State Machine Execution. If an internal transition is triggered, then the corresponding action is executed (there is no change of state and no exit or entry actions must be performed). Otherwise, if an external transition is triggered, we must determine the correct sequence of exit and entry actions to be executed according to the transition source and target state. Transitions outgoing from composite states are inherited from their substates so that a state may be exited because a transition fires that departs from some of its enclosing states. If a transition crosses several state boundaries, several exit and entry actions may be executed in the given order. To this purpose, we seek the innermost composite state that encloses both the source and the target state, i.e. their *least common ancestor*. Then the following actions are executed sequentially: (a) the exit actions of the source state and of any enclosing state up to, but not including, the least common ancestor, innermost first (see macro *exitState*); (b) the action on the transition; (c) the entry actions of the target state and of any enclosing state up to, but not including, the least common ancestor, outermost first (see macro *entryState*); finally (d) the "nature" of the target state is checked and the corresponding operations are performed.

The sequentialization and iteration constructs defined for ASMs in [8] provide the combination of black box – atomic step – view and the white box – durative – view which is needed here to guarantee that when the two ASM rules defined above are executed, all the updates which occur in the macros defined below are performed before the next event is dispatched or becomes releasable. This behavior is reflected by the parameterized macro *stateMachineExecution* (which constitutes the body of the Transition Selection Rule). The macros appearing in this rule are described below.

stateMachineExecution(*trans*) ≡
if *internal*(*trans*) **then** *action*(*trans*)
else seq
exitState(*source*(*trans*),*ToS*)
action(*trans*)
enterState(*FromS*,*target*(*trans*))
case *target*(*trans*)
SequentialState: *enterInitialState*(*target*(*trans*))
ConcurrentState: *startConcurrComput*(*target*(*trans*))
HistoryState: *restoreConfig*(*target*(*trans*))
endcase
where *anc* = *lca*(*source*(*trans*),*target*(*trans*))
ToS = *directSubState*(*anc*,*UpChain*(*source*(*trans*),*anc*))
FromS = *directSubState*(*anc*,*DownChain*(*anc*,*target*(*trans*)))

and *directSubState*: $STATE \times STATE^* \longrightarrow STATE$ is defined by *directSubState*(*s*,*L*) = s' iff $s' \in L$ & *UpState*(s') = s, i.e. s' is the only direct substate of s belonging to the list L.

It remains to define the macros for exiting and entering states, and for the additional actions for sequential, concurrent and history states.

Exiting States. If a transition that crosses the boundary of a composite state fires, we must distinguish two cases to perform the exits from nested states in an order which respects the hierarchical structure (see macro *exitState* below):

1. The agent is not inside a concurrent flow (i.e. *parent*(*Self*) = *undef*). If the agent is (1) not parent of concurrent subagents or (2) it is parent of concurrent subagents but each subagent already performed its exit actions, then for each state from the source state up to, but excluding, the source/target least common ancestor state (see the stateMachineExecution rule above), innermost first, it sequentially (a) stops the internal ongoing activities, (b) performs the exit actions , and (c) removes those states from the agent's current state. Moreover, it (d) updates the history (if defined and provided that the final state has not been reached), memorizing in it all the states it is exiting in case of deep history, or only its direct active substate in case of shallow history, and (e) updates *deferQueue* by deleting all those events which are no more deferred (see macro *sequentialExit* which uses a macro *abortInternalActivity* defined below). In case (2) the agent must furthermore update its *deferQueue* to retain all deferred events of its own but none of those processed by its subagents. Finally it disconnects all its deactivated subagents (see the corresponding macro defined below).
2. The agent is inside a concurrent flow (i.e. *parent*(*Self*) ≠ *undef*). We have to consider the two cases, whether the trigger event is relevant for all the subagents running in parallel within the concurrent state or not. To this purpose we check that the transition source state belongs to the active state of both the agent and its parent (when a subagent is created, it inherits

its parent's current state, therefore at any time the *currState* of the parent agent is a subset of its subagents' *currState*). In this case, each subagent performs the same sequentialExit macro as in the first case, i.e. starting from its deepest state up to, but excluding, its parent's deepest state, it sequentially (a) stops the internal ongoing activities, (b) performs the exit actions and (c) removes those states from the agent's current state. Moreover, it (d) updates the history (if defined and provided that the final state has not been reached) memorizing in it all the states it is exiting in case of deep history, or only its direct active substate in case of shallow history, and (e) updates *deferQueue* by deleting all those events which are no more deferred (see macro *sequentialExit*). Finally, the agent is deactivated, meaning that its rule is set to undef and its current state to the empty set (see macro *deactivate*).
Now consider the case that the transition source state belongs to the active state of at least one but not to all subagents of an agent. Then the event is relevant only for this subagent, and this agent performs the sequential exit as in case 1.

$$
\begin{aligned}
exitState(s,t) \equiv\ & \textbf{if } parent(Self) = undef \\
& \textbf{then if } SubAgent(Self) = \emptyset \\
& \qquad \textbf{then } sequentialExit(s,t) \\
& \qquad \textbf{elseif } noActiveSubAgents \\
& \qquad \textbf{then seq } deferQueue(Self) := defer(deepest(Self)) \cap \bigcap_{a_i \in SubAgent(Self)} deferQueue(a_i) \\
& \qquad\qquad sequentialExit(s,t) \\
& \qquad\qquad disconnectSubAgents \\
& \textbf{if } parent(Self) \neq undef \\
& \textbf{then if } s \in currState(Self)\ \&\ s \in currState(parent(Self)) \\
& \qquad \textbf{then} \\
& \qquad\qquad sequentialExit(S, S') \\
& \qquad\qquad deactivate(Self) \\
& \qquad \textbf{else } sequentialExit(s,t) \\
& \\
& where\ S = deepest(Self) \\
& \qquad S' = deepest(parent(Self)) \\
& \qquad noActiveSubAgents = \forall a_i \in SubAgent(Self) : currState(a_i) = \emptyset
\end{aligned}
$$

For the definition of the macro *sequentialExit* we use the macro *abortInternalActivity* which will be defined below. In defining *sequentialExit* we use a function $hist(s, S)$ whose value depends on whether S is a deep history state or not. $hist(s, S)$ yields $UpChain(s,\widehat{S})$ for deep history, $directSubState(S, UpChain(s,S))$ for shallow history.

```
sequentialExit(s,t) ≡ loop through S ∈ UpChain(s,t)
                        seq
                            abortInternalActivity(S)
                            exit(S)
                            currState := remove(S,currState)
                        endseq
                        if history(S) ≠ undef & final(S) ∉ currState
                        then memory(history(S)) := hist(s, S)
                      endloop
```

```
deactivate(a) ≡ Rule(a) := undef
                currState(a) := ∅
```

```
disconnectSubAgents ≡ do forall a_i ∈ SubAgent(Self)
                                      parent(a_i) = undef
```

Entering States. A transition may have a target state nested at any depth in a composite state. Therefore, any state enclosing the target one up to, but excluding, the least common ancestor will be entered in sequence, outermost first. Entering a state means that (a) the state is activated, i.e. inserted in *currState*, (b) its entry action is performed, and (c) the state internal activity (if any) is started. This is realized by the macro *enterState* for which we use the macro *startActivity* defined below. The agent's *deferQueue* is updated by deleting all those events which are no more deferred in the target state.

```
enterState(s,t) ≡ loop through S ∈ DownChain(s,t)
                    seq
                        currState := insert(S,currState)
                        entry(S)
                        startActivity(S)
                    endseq
                    deferQueue := deferQueue ∩ defer(S)
                  endloop
```

Internal Activities. When a state is active, its internal activity (if any) is required to be executed. Apparently, internal activities are intended as concurrent and [4] imposes no particular scheduling conditions for them. We model this by creating a new *worker* agent whose job is to execute the activity of its associated state. The *worker* agent is created when the state is entered and after its entry action has been executed. It receives as program the ASM $A(S)$ formalizing the state activity.

```
startActivity(S) ≡ extend AGENT with a
                     Rule(a) := A(S)
                     worker(S) := a
```

Using an ASM as rigorous replacement for the intuitive notion of "internal UML activity", we obtain a mathematically rigorous definition without loosing generality[10]. In addition we make the UML notion of "ongoing" activity precise by defining it as steps of an ASM in a multi-agent distributed ASM run.

If an activity is aborted prior to its termination as result of the firing of an outgoing transition, then before leaving the state its associated worker agent is deactivated since its job is terminated. This is performed by the following macro which is used for defining *sequentialExit*.

$$\begin{aligned} abortInternalActivity(S) \equiv\ & Rule(worker(S)) := undef \\ & worker(S) := undef \end{aligned}$$

Sequential Composite States. A transition drawn to the boundary of a sequential composite state is equivalent to a transition to its initial pseudostate [4]. Therefore, when a composite sequential state is the target state of a triggered transition, the control passes to its initial state that is inserted in *currState*.

$$enterInitialState(S) \equiv currState := insert(init(S), currState)$$

History States. If a transition incoming to a history state within a composite state fires, the configuration of active states stored in its *memory* is restored. Therefore, each state in the history is activated, i.e. it is inserted in *currState*, its entry action is performed, its activity is executed, and the state is removed from the history. The right entering order is guaranteed by the LIFO structure of *memory*.

Observe that when a state is entered for the first time or its most recently active state prior to its exit was the final state, its history (if any) must be empty [4]. This is guaranteed in our model since we initialize each history state memory to the empty sequence, delete it after using it, and store nothing in it when its enclosing state is exited by a final state.

$$\begin{aligned} restoreConfig(H) \equiv\ & \textbf{loop through } S \in memory(H) \\ & \quad \textbf{seq} \\ & \quad\quad currState := insert(S, currState) \\ & \quad\quad entry(S) \\ & \quad\quad startActivity(S) \\ & \quad\quad memory(H) := delete(S, memory(H)) \\ & \quad \textbf{endseq} \\ & \quad deferQueue := deferQueue \cap defer(S) \\ & \textbf{endloop} \end{aligned}$$

Concurrent Composite States. If a transition incoming to a concurrent composite state fires, the flow of control is split into two or more flows of control. The currently active agent creates a new agent a_i for each concurrent component S_i. All the subagents inherit their parent's program to execute statechart diagrams,

[10] That no generality is lost derives from Gurevich's proof of the ASM thesis in [13].

its *currState* configuration and the parent's list of active deferred events. As a transition drawn to the boundary of a concurrent composite state is equivalent to a transition to any of its concurrent components and consequently to the component initial state, each agent a_i activates the component S_i and its associated initial state.

startConcurrComput(S) ≡
  let $S_1, \ldots, S_n$ = *concurrentComp*(S)
    extend *AGENT* **with** $a_1, \ldots, a_n$
      do forall $1 \leq i \leq n$
        parent(a_i) := *Self*
        deferQueue(a_i) := *deferQueue*(*Self*)
        Rule(a_i) := *Rule*(*Self*)
        currState(a_i) := *insert*({S_i,*init*(S_i)},*currState*)

The parent agent will stand idle waiting for the termination of its subagents' computation. This is enforced by the definition of when a concurrent state is completed to trigger the completion event which may enable the transition exiting the concurrent state. The running subagents can finish their job either because of a completion event generated by their parent[11] or by the firing of an explicit event labeling a transition outgoing their enclosing state. In our model the substates' exit action and internal activity abortion are performed by the *exitState* macro, in a synchronized fashion. Other choices are easily defined modeling our rules appropriately. The UML documents seem not to mention this semantically relevant issue at all.

Remark. In a concurrent compound state S', a transition *trans*(*e,g,a*) outgoing from a state S in a concurrent component and incoming to a state S'' sibling of S' (see Fig. 1.a), can be viewed as split into two transitions (see Fig. 1.b): a transition *trans*(*e,g,Send exitEvent*(S)) from S to S', where *Send exitEvent*(S) is an event generation action[12], and a transition *trans*(*exitEvent*(S),*true,a*) from S' to S''. To guarantee the expected semantics of UML statecharts, we impose,

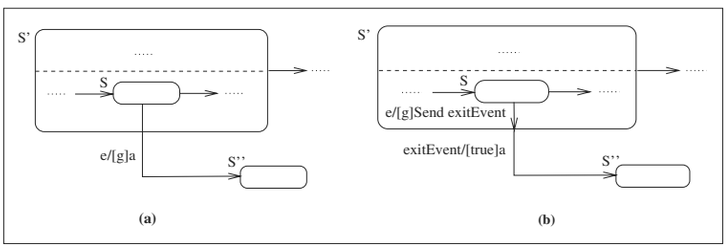


Fig. 1.

[11] In this case they all must have reached their final state.
[12] According to [4] an action labeling a transition may consist in sending a signal.

as an integrity constraint on the permissible runs, that the event *exitEvent*(S) must be dispatched before any other event (see the event handling mechanism in section 3.4).

4 Semantical Equivalence among Building Blocks

UML statecharts encompass for notational convenience some constructs which can be defined in terms of basic constructs. Not to overload our model, we decided to include only the basic notations and to sketch here how to replace the remaining constructs by combinations of basic constructs.

Fork-Join Pseudostates. *Fork* and *join* pseudostates split and merge transitions arriving at, emanating from, concurrent states. A transition to the boundary of a concurrent compound state is equivalent to a transition to each of its direct substates (and therefore to their initial states), and a transition from the boundary of a concurrent compound state is equivalent to a transition from the final states of each of its substates. Therefore, the fork (resp. join) semantics can be obtained by allowing only incoming (resp. outgoing) transitions that terminate on (resp. depart from) the boundary of concurrent states, and imposing that each concurrent substate must enclose an initial and a final state.

Junction Pseudostate. *Junction* states are used only to chain together multiple transitions – this is known as *merge* –, or to split an incoming transition into multiple outgoing transitions labeled with different guard conditions – this is known as *conditional branch* [4].

Submachine States. UML statecharts provide also *submachine states*, a syntactical convenience to facilitate reuse and modularity [4]. A submachine state is only a shorthand that implies a macro-like expansion by another state machine and is semantically equivalent to a composite state. According to the UML metamodel, a submachine state is of the form *state*(*entry,exit,do*(A)*,include*(S')). One can assume that each occurrence of a submachine state is substituted by the sequential[13] composite state defined by *entry, exit, do*(A)*, S'*. Moreover, we identify transitions directly incoming to, respectively outgoing from, the submachine state with transitions directly incoming to, respectively outgoing from, the resulting sequential composite state.

Stub States. A stub state is nothing else then an alias for an entry point to or an exit point from a state s in S' of a submachine state *state*(*entry,exit,do*(A)*,include*(S')).

Additional Constructs. *Synch states* are used to synchronize the execution of concurrent substates. Their semantics can be given by slightly modifying the above formalization of concurrent states.

[13] Submachine states are never concurrent [4].

5 Conclusion and Related Work

In this section we discuss some ambiguities in the official semantics of UML [4,5, 20] which are resolved in the ASM model. We also show how UML requirements for state machines are satisfied by our model.

The state machine execution is formalized through the macro *stateMachine-Execution* (invoked by the rule Transition Selection) that reflects the scheme of a generic control machine. These ASM statecharts generalize the *Mealy ASMs* defined in [6].

Our model reflects all the characteristics of the state machines metamodel in [4] and adds to its structural, static definition the underlying control flow semantics. A subtle question regards the execution of ongoing state activities. What does happen when an internal transition occurs? Does the activity interrupt and then restart from the same computation point, or does it never interrupt? The way we model internal activities guarantees the second, to our understanding reasonable, alternative. However, our model can be easily adapted to formalize other behaviors.

By replacing the undefined UML terms of "action" and "activity" with (possibly structured, in the sense of [8]) "ASM rule", we provide a precise mathematical content to these terms without loosing the generality intended by the designers of UML (see in this connection Gurevich's proof of the ASM thesis [13]). Our model also provides a precise meaning of the vague UML term "ongoing internal activity", namely as execution of an ASM in a multi-agent distributed run as defined in [12]. The sequentialization, iteration and submachine constructs defined for ASMs in [8] clarify in what sense sequences of nested exit and entry actions can be guaranteed to be executed in one "run to completion step", as postulated by the UML documents, namely before the next event may trigger the next "step". Our model also makes some semantically relevant features[14] explicit which seem not to have been considered in the official UML documents.

Several semantics for statecharts have been proposed in the literature [21]. Most of these are concerned with modeling Harel's statecharts, whose semantics is rather different from UML state machines (e.g. in the event handling policy). Although our model can be adapted to grasp such differences, our intent is to define the UML state machine semantics up to the degree of precision one can reach without compromising the desired freedom of the so called "semantic variation points".

Differently from the formalization of UML state machines in [11,19], our model reflects the original structure of machines as described in the UML documents, without imposing any graphical transformation or flattening of diagrams. [11] uses graph rewriting techniques to transform UML state machines into a "normal form" machine, without considering the execution of actions and activities. The model in [19] leaves out some state machines features, and some are covered by means of semantical equivalences which, however, do not

[14] E.g. whether abortion of internal activities and exit actions of concurrent agents should be synchronized or not.

always respect the UML metamodel constraints (see [4], pp. 2-126). For instance, entry/exit actions in a state are replaced by attaching such actions respectively to the state incoming/outgoing transitions, whereas the metamodel asserts that the multiplicity of `Action` in `Transition` is 0..1, that is no or exactly one action may label a transition.

In [17] the operational behavior of UML state machine constructs is described using pseudo-code in a way which in many places includes specific implementation decisions (mostly without stating them), whereas we tried to let the intended semantic variation points of UML stand out explicitly as such.

References

1. Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.
2. *Rational Software Corporation, Unified Modeling Language UML, version 1.3*, 1999.
3. *UML 1.3 Notation*, 1999. (Published as part of [2]).
4. *UML 1.3 Semantics*, 1999. (Published as part of [2]).
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
6. E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
7. E. Börger, A. Cavarra, and E. Riccobene. An ASM Semantics for UML Activity Diagrams. In T. Rus, editor, *AMAST2000*, volume 1816 of *LNCS*, pages 293–308. Springer Verlag, May 2000.
8. E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In *Gurevich Festschrift CSL 2000*, 2000. (To appear).
9. A. S. Evans, J-M. Bruel, K. Lano R. France, and B. Rumpe. Making UML Precise. In *In OOPSLA'98 Workshop on Formalizing UML. Why and How?*, October 1998.
10. R. B. France, A. S. Evans, K. C. Lano, and B. Rumpe. Developing the UML as a formal modeling notation. *Computer Standards and Interfaces: Special Issues on Formal Development Techniques*, Accepted for publication, 1998.
11. Martin Gogolla and Francesco 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.
12. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
13. Y. Gurevich. Sequential Abstract State Machines capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1, 2000. (To appear).
14. D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer, IEEE Computer Society*, 30(7):31–42, 1997.
15. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Trans. Soft. Eng. method*, 5(4):293–333, 1996.
16. D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts*. McGraw-Hill, 1998.

17. Ivan Paltor and Johan Lilius. Formalising UML state machines for model checking. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*. Springer, 1999.
18. The precise UML group. `http://www.cs.york.ac.uk/puml/`.
19. 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*. Lecture Notes in Computer Science, 2000. (To appear).
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
21. M. von der Beek. A Comparison of Statechart Variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 526. Lecture Notes in Computer Science, 1994.

On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine

Robert Eschbach[1], Uwe Glässer[2], Reinhard Gotzhein[1], and Andreas Prinz[3]

[1] Department of Computing Science, University of Kaiserslautern, D-67653 Kaiserslautern, Germany,

[2] Heinz Nixdorf Institute, University of Paderborn, D-33102 Paderborn, Germany,

[3] Department of Computer Science, Humboldt-University Berlin, D-12489 Berlin, Germany

Abstract. In November 1999, a new version of *SDL (Specification and Description Language)* called *SDL-2000* has passed ITU, an international standardization body for telecommunication. SDL is a fairly complex, graphical formal description technique for the development of distributed systems, and has been broadly used in industry for many years. Efforts to define the semantics of SDL-2000 formally have started early in 1998. By now, a draft formal semantics is available, which is determined to become the official formal SDL semantics after its approval in 2000. It is based on the formalism of *Abstract State Machines (ASMs)*, which has been selected for several reasons including intelligibility and executability.
The formal semantics of SDL addresses the static semantics, transformation rules, and the dynamic semantics. The approach taken to define the dynamic semantics is particularly interesting. Although basically being operational, it differs from existing approaches in several ways. In this paper, we address and highlight some of these differences, using a simplified specification language called SSL instead of SDL. In defining a formal dynamic semantics for SSL, we formally describe an abstract machine, a compilation function mapping SSL specifications to code of this machine, and an operational definition of the set of initial states, using ASM as the underlying formalism. Furthermore, we present in some detail the semantics of SSL procedure calls.

1 Introduction

Since 1976, SDL (Specification and Description Language) is an ITU standardized language[1] for the development of distributed real-time systems in general, and telecommunication systems in particular. With its graphical syntax, its object-oriented features, its direct support for reuse, and its integration with

[1] SDL is defined by the ITU-T, the Telecommunication Standardization Sector of the International Telecommunication Union (ITU), in their *Recommendation Z.100* [20].

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 242–265, 2000.

other development languages, SDL satisfies the primary needs of system developers. Today, SDL is being broadly applied in industry, and is being supported by powerful commercial tool environments.

In 1988, the semantics of SDL has been defined formally, upgrading the language to a formal description technique (FDT). This formal definition has been updated and maintained for subsequent versions of the SDL language in 1992 and 1996. Building on a combination of the VDM meta language Meta-IV with a CSP-like communication mechanism, the formal semantic model provides a comprehensive formalization of all static and dynamic aspects of SDL, though, it is hardly manageable because of its size[2]. Essentially, the formal semantics is given by a set of Meta-IV programs that take an SDL specification as input, determine the correctness of its static semantics, perform a number of transformations to replace non-basic language constructs, and interpret the specification. It has been argued that this style of defining the formal semantics is particularly suitable for tool builders.

In November 1999, a new version of the language, called SDL-2000, has been approved as international standard. In addition to the usual language maintenance, SDL-2000 offers important new features, including object-oriented data type definitions, a unified concept for blocks, processes, and composite states, and exception handling. Based on the assessment that the existing Meta-IV programs would be too difficult to update and maintain, it was decided to conceive a new formal semantics for SDL-2000 from scratch. For this purpose, a special task force, the *SDL semantics group*, consisting of experts from Germany and China including the authors of this paper, was formed. The draft formal semantics defined by this group (see [24]) has been determined to become the official formal SDL semantics after its approval, which is expected for November 2000.

Among the primary design objectives of this approach is *intelligibility* and conciseness of the formal description as a prerequisite for acceptance, correctness and maintainability. Additionally, *executability* has become a key issue. Executability calls for an operational formalism with readily available tool support. For this and other reasons (cf. [11]), *Abstract State Machines (ASMs)* have finally been selected as the underlying formalism. To support executability, the formal semantics defines, for each SDL specification, corresponding ASM code. This differs substantially from the interpreter view taken in previous work, and enables SDL-to-ASM-compilers. Thus, given the availability of an SDL-to-ASM-compiler, it will be straightforward to generate reference implementations.

The approach taken by the SDL semantics group to formalize the dynamic semantics of SDL-2000, though basically operational, differs in several aspects from the approach taken to define the semantics of SDL-88 through SDL-96, and also from the style of other ASM approaches (cf. [12,10]):

- The style of the semantics has the characteristics of a *compiler approach*, as compared to the interpreter approach chosen, for instance, to define the

[2] The complete formal semantics for SDL-96, as defined in Annex F to Z.100, is documented on more than 500 pages of Meta-IV descriptions.

semantics of SDL-88 through SDL-96. A compiler usually performs lexical and syntactic analysis functions, just like an interpreter does. Unlike an interpreter, compilers produce executables for some kind of machine, instead of executing a version of the source directly. Thus, the source is viewed as a collection of instructions rather than a data structure.

- In practice, processing specifications into an abstract syntax representation is simpler and faster than compiling it into some kind of executable code. On the other hand, compilation into code directly enables executability. There is a technique to combine the advantages of both approaches, namely: the generation of abstract code "running" on an abstract machine. The SDL semantics group has devised an *SDL Abstract Machine*, SAM for short, which is defined in terms of a distributed real-time ASM. SDL specifications are then compiled into the instruction set of this machine.
- Compared to the standard technique for defining the set of initial states of an ASM model—usually this is done in an axiomatic or declarative style[3]—a realistic semantic model of SDL requires a different approach. Following the view of Z.100 on how initial system states are derived from given SDL specifications, the SAM model starts its execution from a set of so-called *pre-initial* states that are canonically defined. The actual set of initial states is then determined in an operational way as result of an *initialization phase*. This way, the initialization phase and the execution phase may even overlap.

Due to the complexity of SDL-2000, the formal semantics is of substantial size. Therefore, in order to present and illustrate the application of the basic techniques above to the definition of the SDL semantics, we have chosen a very small language called *SSL (Simple Specification Language)*. In the following, we define an abstract machine for SSL, a compilation function mapping SSL specifications to code running on this machine, and the initialization and execution phases. Moreover, we present in more detail the semantics of SSL procedure calls. Our model of procedure semantics allows for call-by-value and call-by-reference semantics and supports the concept of static binding.

Taking into account that the work reported here is not an academic exercise, but takes place in a real-life industrial setting, pragmatic compromises are unavoidable. For this reason, some decisions taken in the formalization of the SDL (and hence SSL) semantics are rather pragmatic or strongly influenced by the SDL experts. In our opinion, such work is crucial to demonstrate the usefulness of formalisms such as ASMs. In this sense, the paper reports on work that is still in progress, but which has reached a stage where most "design" decisions have been taken, and a sufficiently stable draft is available (cf. [24]). A preliminary version of this paper can be found in [7].

This paper is organized as follows. Section 2 gives an overview on related work. Pragmatic aspects of our formalization approach are briefly discussed in Section 3, while Section 4 provides an overview of the formal Semantics for SDL-2000.

[3] Quite often, an initial state S_0 is defined through a set of first-order formulas $\varphi_1, \ldots, \varphi_n$ such that $S_0 \models \varphi_1 \wedge \ldots \wedge \varphi_n$.

Section 5 introduces the language SSL. Section 6 defines the abstract compilation of SSL specifications to SSLM programs. Section 7 defines the SSL abstract machine (SSLM). Section 8 describes the initialization. Section 9 presents conclusions. Throughout this paper we assume the reader to be familiar with Gurevich's notion of ASM (cf. [14,15,16]).

2 Related Work

Regarding the importance of SDL as a practical tool for industrial systems engineering, it is certainly not astonishing that in addition to activities within ITU, there is a considerable variety of attempts to formalize the semantics of SDL using various formal methods. According to the principal objectives behind these attempts, one can distinguish two basically different aspects, namely: (1) approaches that are mainly concerned with analysis and verification of SDL system specifications; (2) approaches that are mainly concerned with formal documentation, maintenance and validation of the SDL language definition.

2.1 Analysis and Verification

In [1], Bergstra and Middleburg define a *process algebra* semantics of a restricted version of SDL, which they call φSDL (where φ stands for "flat"), as it does not cover the structural aspects of SDL. Additionally, they introduce a restricted notion of time to simplify time related concepts of SDL. The authors claim to have convincing pragmatic justification for their choices; for instance, they argue that *"a dramatically simplified version of SDL"* and an adequate semantics for it *"are prerequisites for advanced analysis and formal verification"*. After all, they deploy an impressive mathematical apparatus making it difficult to see whether φSDL is relevant in practice.

Broy [4], Holz and Stølen [19], and Hinkel [18] model various subsets of (essentially) Basic SDL using denotational semantics based on *stream processing functions* of *FOCUS* [5]. While it may be natural to model SDL process communication as discrete streams of signals, the functional view does neither support the concept of system states and state transitions nor allows the stream formalism for an adequate treatment of time. Even the most comprehensive model [18] builds on a fundamentally restricted notion of *global system time* not allowing to express time quantities explicitly. In particular, it is not possible to specify any quantitative delays as required for dealing with the timer concept of SDL. Since the expiration time of a timer does not refer to a global system time (as represented by the SDL expression **now**), the meaning of an active timer is reduced to the fact that a timer signal will eventually be generated (after some finite delay). This results in a severe oversimplification.[4]

[4] It is certainly right that the current definition of SDL needs further improvements allowing a more detailed and more precise inspection of timing behavior (see for instance [22]), e.g. to determine delays caused by time consuming operations; on the other hand, any reasonable approach to overcome these deficiencies cannot mean to replace the current time concept by an even more primitive one, as done in [18].

Fischer, Dimitrov and Tauber propose an extended Petri Net model, so-called *SDL Time Nets*, as a formal basis to verify SDL protocol specifications [8], [9]. The transformation of SDL specifications into corresponding net models as well as the transformation of results back to the SDL level is done automatically within the SDL Integrated Tools Environment *SITE*. Though these transformations, in principle, remain invisible for the user, the authors concede that certain knowledge is necessary about the internal data structures of the analysis tools. Moreover, the integration of further net analysis algorithms is difficult and requires detailed knowledge about the internals (like interfaces, structure and representation formats) of the tools.

2.2 Documentation and Validation

An ambitious attempt aiming at a comprehensive formal semantics for SDL-92 as a basis for further development of SDL was started by Lau and Prinz with their definition of BSDL (*Base SDL*) [21]. With the ultimate goal to simplify the language definition, they proposed a modeling approach using *Object-Z* to define a universal *core* for SDL as a conceptual framework to deal with the main building blocks of the language. This core should be as clear and concise as possible, yet it should include the basic model of processes and communication, all the object-oriented concepts and partly also structuring concepts. The meaning of additional language constructs (not handled within the core) should then be fixed by defining their transformation to the core. In that respect, the BSDL approach is similar to the original formal model of SDL, but suggests a more systematic and robust construction of a formal semantic model. However, the Object-Z definition of BSDL was rejected by the ITU even before the work was completed.

Regarding further developments of SDL towards SDL-2000, an approach based on *transition systems* generated from attributed abstract syntax trees (ASTs) is outlined in [13]. An attribute grammar is formed by adding evaluation rules to the abstract grammar of the language, where attributes for instance may represent *actions*. This way, one obtains for each complete SDL specification an attributed AST containing all the information required to generate a behavior model. The underlying mathematical model and the notation are not determined, which leaves room for subsequent choices.

In [12] and [10], an ASM behavior model for Basic SDL-92 is defined. Starting from an abstract operational view, the dynamic properties of Basic SDL-92 are formalized in terms of an *abstract SDL interpreter* based on a *distributed real-time ASM* model. This work provides a conceptual framework that has further been developed and extended by combining it with the compiler-based approach from [13], as well as certain BSDL concepts from [21], resulting in a robust formal basis for the definition of an ASM behavior model for SDL-2000, as discussed here.

3 Pragmatic Foundation

This section addresses fundamental questions concerning the practicability, suitability and robustness of our formalization approach. The primary focus here is on pragmatic aspects rather than on technical details. That is, we try to explain at a conceptual level distinctive features of our SDL behavior model. In particular, we try to answer the question, "Why and how does the abstract operational view of the ASM method in combination with the compiler approach practiced here naturally meet the demands on formal documentation, language maintenance and tool development as arising from the SDL standardization process?".
Industrial engineering of complex distributed systems based on SDL, as already outlined in Sect. 1, has quite a tradition in telecommunication technology. Meanwhile, one can as well observe an increasing proliferation of SDL applications in various fields of information technology (e.g., engineering of distributed control systems). Like the development of SDL over its life cycle, this progress has been influenced by the apparent convergence of information technology and communication technology.
To meet the needs of system design experts, the language has been continuously improved over a period of more than 20 years, evolving from a primitive graphical notation to a sophisticated formal description technique, according to the demands on industrial systems technology. Consequently, the underlying language concepts closely reflect the view and the intuitive understanding of system behavior and system structure of those who have mainly contributed (through their practical experience) to the development of SDL (cf. the standard literature, e.g. [23,6]). Hence, in order to gain acceptance, any realistic attempt to formalize the semantics of SDL with the intention to facilitate language design and validation of implementations therefore cannot succeed without supporting the intuitive understanding of well established SDL concepts (including functional, structural and timing aspects) as defined in Z.100.

3.1 Abstract Operational View

The problem of "turning English into mathematics" is considerably simplified by a close correspondence between the specification method and the semantic model to be formalized. In fact, this allows for more natural and adequate abstractions resulting in more comprehensible and more reliable descriptions.[5] Given the definition of SDL in Z.100, it makes perfect sense to formalize dynamic properties of SDL specifications as abstract machine runs since this style of modeling directly reflects the abstract operational view of the informal language description. Conceptually, the effect of actions and events as associated with *active* SDL objects (i.e., those objects that have a behavior) on the global system state is stated in terms of the effect of SAM operations on abstract machine states.

[5] For instance, was it possible to eliminate a number of cumbersome in-language transformations (specified by so-called *Model Sections*) from Z.100 by defining the meaning of certain non-basic SDL constructs directly in the formal behavior model.

More specifically, the resulting abstraction depends on the *granularity* of abstract machine operations specified by the SAM instruction set. Ideally, this granularity is such that all relevant effects on the system state are highlighted, whereas minor operational details remain hidden in the functions and predicates on top of which these operations are formulated. Additional details may then be introduced through stepwise refinements such that the style of representation gradually turns from declarative to operational while crossing abstraction levels.[6]
Essential for a direct and concise encoding of SDL behavior primitives in ASM notation is of course a close correspondence of the underlying computation models. In that respect, there are two observations concerning our modeling approach:

- The SDL view of distributed systems with real-time constraints and the semantic modeling concept of distributed real-time ASM clearly coincide. That is, essential properties of the underlying computation models – namely, the notions of concurrency, reactivity and time as well as the notion of state – are so tightly related that the common understanding of SDL can directly be converted into a formal semantic model avoiding any formalization overhead.
- Even without direct support of object-oriented features in ASMs, the resulting ASM model of the dynamic semantics is particularly concise, readable and understandable. Furthermore, this model can easily be extended and modified as required for an evolving technical standard. Beyond the purpose addressed here, the model may as well be utilized for defining a *bridging semantics* in order to combine SDL with other modeling languages (e.g., UML and VHDL).

3.2 Correctness and Reliability

Defining semantics by a mapping to a well defined semantic basis, as realized by the compilation of SDL specifications into SAM code, is a well-known technique, but is often considered to be as dangerous as helpful. Of course one may argue that this leads to the old "the compiler defines the language semantics" problem: "Who validates and/or verifies the correctness of this mapping and with respect to which origin?".
Now, the question here is: "How can one establish that a *formal* semantic model like the ASM behavior model of SDL faithfully reflects the intuitive understanding of SDL, i.e. yields a valid interpretation of the definitions in the language reference manual?". Since there is no way of proving correctness (in a strict mathematical sense), the approach taken here is to construct an ASM behavior

[6] An illustrative example is the abstract definition of the behavior model for SDL channels (see [24]). By exploiting the decentralized organization of the underlying signal flow model, the entire static information on the system structure and the reachability constraints are completely encapsulated into a predicate *compatible*, which can then be refined by introducing technical details on how this information is to be derived from a given specification.

model that reflects the dynamic properties of the language so closely that correctness can be established by observation and experimentation. In ASM terms, such a model is called a *ground model*, cf. [2]. The quality of ground models considerably depends on the underlying abstractions for dealing with real-world phenomena, i.e. on the way in which basic objects and operations of a ground model are related to basic entities and actions as observed in the real world.
Investigating the construction of ground models (as part of the analysis and design process), one can identify general principles for justifying their appropriateness and validity. In [2], Egon Börger convincingly argues that this justification process has three basic dimensions, namely: a conceptual justification, an experimental justification, and a mathematical justification. According to this understanding, the best we can achieve is a solid pragmatic foundation.

4 Overview of the Formal SDL Semantics

The definition of the formal semantics of SDL-2000 is structured into the following major parts:

- grammar,
- well-formedness conditions (static semantics),
- transformation rules, and
- dynamic semantics.

The *grammar* defines the set of syntactically correct SDL specifications. In the SDL standard, a concrete textual, a concrete graphical, and an abstract grammar are defined using BNF with some extensions to capture graphical language elements. The abstract grammar is obtained from the concrete grammars by removing irrelevant details such as separators and lexical rules.
The *well-formedness conditions* impose additional context constraints on syntactically correct specifications, such as which names it is allowed to use at a given place, which kind of values to assign to variables, etc.
For some language constructs, the formal semantics is not provided directly, but through *transformation rules*. These rules define how a given specification is to be transformed in order to replace the language constructs by basic language elements, and are formally represented as rewrite rules.
Finally, the *dynamic semantics* is given to syntactically correct SDL specifications satisfying the well-formedness conditions, after application of the transformation rules. The dynamic semantics consists of the following parts as illustrated by figure 1.

- The *SDL Abstract Machine (SAM)* defines basic signal flow concepts of SDL such as signals, timers, exceptions, and gates in terms of the ASM model. Furthermore, ASM agents are specialized to model agents in the context of SDL. Finally, several signal processing and behavior primitives - in a sense the abstract machine instructions of the SAM - are predefined.

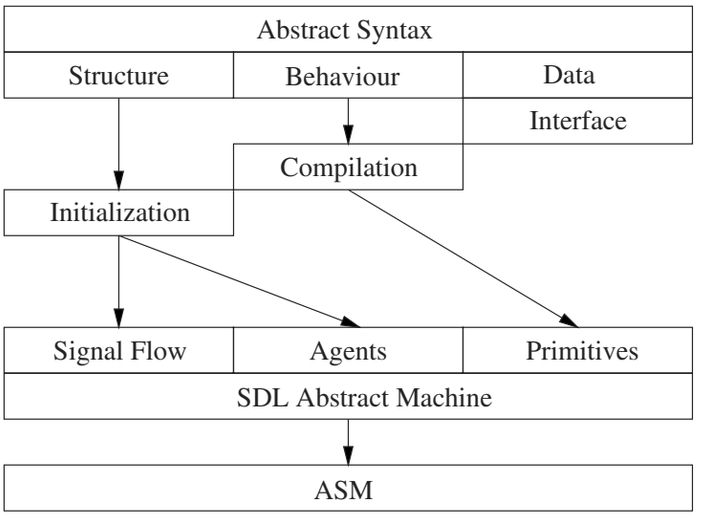


Fig. 1. Overview of the Dynamic Semantics

- An *SDL Abstract Machine Program* defines a set of computations. Computations consist of an initialization phase and an execution phase. SAM programs have predefined parts that are the same for all SDL specifications, and variable parts that are generated from the abstract syntax representation of a given SDL specification.
  - The *initialization* handles static structural properties of the specification by recursively unfolding all the static objects of the specification. In fact, the same happens during the execution phase when new SDL agents are created dynamically. From this point of view, the initialization is merely the instantiation of the SDL system agent.
  - The *compilation function* maps behavior representations into the SAM primitives. This function amounts to an abstract compiler taking the abstract syntax tree (AST) of an SDL specification as input and transforming it to the abstract machine instructions.
- The *data semantics* is separated from the rest of the dynamic semantics by an interface. The use of an interface is intentional at this place. It will allow to exchange the data model, if for some domain another data model is more appropriate than the SDL built-in model. Moreover, also the SDL built-in model can be changed this way without affecting the rest of the semantics, thus enhancing its maintainability.

As in the past, the new formal semantics is defined starting from the abstract syntax of SDL, which is documented in the SDL standard. From this abstract syntax, a behavior model that can be understood as abstract code generated

from an SDL specification is derived. The dynamic semantics associates, with each SDL specification, a particular distributed, real-time ASM. Intuitively, an ASM consists of a set of autonomous agents cooperatively performing concurrent machine runs. This approach differs substantially from the interpreter view taken in the previous SDL semantics definition, and will enable SDL-to-ASM compilers. Conceptually, one can identify three basic layers at which our semantic model of SDL is actually defined (either explicitly or implicitly), namely:

(2) the compilation function,
(1) the SAM architecture, and
(0) the semantic model of distributed real-time ASM.

Regarding layers (2),(1), it is not a priori fixed which aspects of behavior are subject to modeling at which one of these two layers. Though, in principle, specific aspects depending on a given SDL specification typically ought to be encoded into the compilation function, whereas invariant dynamic properties are more naturally expressed in terms of the SAM model. Finally, to keep the formalization as concise and elegant as possible, all aspects that both computation models (SDL and ASM) have in common are preferably given implicitly by mapping SDL behavior primitives directly onto corresponding primitives of the underlying ASM model at layer (0).
By splitting the definition of the SDL behavior model into the above layers, one obtains a flexible framework for semantic modeling that allows for natural and adequate abstractions, transparent and well defined formalization, and flexible rearrangements on the basis of an executable semantics.

5 SSL: A Simple Specification Language

In order to illustrate our approach to the formal SDL-2000 semantics and the application of the basic techniques introduced in section 1 we have chosen a very small language called SSL (Simple Specification Language). SSL is untyped, and is based on a small number of statements including assignment and the conditional statement. Furthermore, an SSL-Specification may contain procedure definition and procedure calls. A procedure definition can contain reference parameters and value parameters.

5.1 Basics

The syntax of SSL is given below. We use Name as a predefined lexical unit and describe the syntax using a variant of BNF (Backus-Naur-Form). More precisely we will just give an *abstract syntax* of the language. We assume the concrete syntax to be handled by an ordinary parser that provides us with the abstract representation in terms of the abstract syntax.
Please note, that we use prefixes in the syntax to make the role of the items clear. Prefixes are separated from the nonterminal name by a "-". Prefixes do not have a semantics.

5.2 Specifications and Procedures

An SSL specification describes a concurrent system. It is given as a list of variables, a list of local specifications, a list of locally defined procedures, and a main expression. To execute an SSL specification means to first start the execution of the enclosed SSL specifications and then to evaluate the main-expression. The enclosed specifications evaluate their main-expression concurrently with.

$$\begin{aligned} \text{sslSpec} ::= {} & \text{spec-Name variableDecl}^* \text{ sslSpec}^* \text{ procedureDef}^* \\ & \text{main-expression} \\ \text{variableDecl} ::= {} & \text{Name} \end{aligned}$$

```
specification Hanoi
procedure hanoi (ref x,y,z, val n)
  if n = 1 then
     x := x - 1; z := z + 1; return;
  else
     call hanoi(x,z,y,n-1); call hanoi(x,y,z,1);
     call hanoi(y,x,z,n-1); return;
end hanoi

var x,y,z,n
begin Hanoi
  x := 4; y := 0; z := 0; n := 4; call hanoi(x,y,z,n);
end Hanoi
```

Fig. 2. Specification Hanoi

A procedure is given as a list of its parameters and an expression for the body.

$$\begin{aligned} \text{procedureDef} ::= {} & \text{proc-Name refParam}^* \text{ valParam}^* \text{ variableDecl}^* \\ & \text{procedureDef}^* \text{ main-expression} \\ \text{refParam} ::= {} & \text{Name} \\ \text{valParam} ::= {} & \text{Name} \end{aligned}$$

5.3 Expressions

Expressions in the scope of SSL include also the usual behavior of statements. In this case, their return value is simply ignored. The nonterminal function denotes constants (like 0 and 1) as well as functions (as + and -). The meaning of these constructs is supposed to be part of the language. The semantics of ifExpr, variable, assignment, procCall, procReturn and equality is as usual. For sequence,

the semantics is the sequential interpretation of the subexpressions. The resulting value of the sequence is the last one.

expression ::= function | ifExpr | variable | equality | sequence | assignment | procCall | procReturn
function ::= Name expression*
ifExpr ::= test-expression then-expression [else-expression]
variable ::= Name
equality ::= lhs-expression rhs-expression
sequence ::= fst-expression nxt-expression
assignment ::= Name expression
procCall ::= Name ref-Name* expression*
procReturn ::= expression

Please note, that the syntax domain names are also used as the constructors for the syntax tree. A sample SSL specification can be found in figure 2.

6 Compilation

The compilation defines a mapping from the behavioral part of the abstract syntax into ASM behavior. In fact, an abstract code is defined as the result of this mapping. The behavior belonging to the behavior primitives in the abstract code is defined within section 7.

6.1 Basics

In this section, we define a compilation function for SSL specifications and for SSL procedures.

$compileSpec$: sslSpecification $\rightarrow$ $AbstractCode$
$compileProc$: procedureDef $\rightarrow$ $AbstractCode$

For the representation of code, we use a special labeling. Each instruction has a label attached to it. The specification keeps the information of the current label and selects the instruction to process according to this label. Therefore, we introduce a static domain *Label*. The abstract code is then a set of instructions and an instruction a tuple consisting of a label and a primitive.

$AbstractCode = Instruction$**-set**
$Instruction = Label \times Primitive$.

The compiled code of all specifications and procedures is merged together in the initial state. Specifications and procedures are characterized by the functions

$entrySpec$: sslSpecification $\rightarrow$ *Label*,
$entryProc$: Name $\rightarrow$ *Label*,

which return the entry labels. For SSL we introduce the following primitives. Again, the prefixes are without semantic relevance.

$$
\begin{aligned}
\mathit{Primitive} &= \mathit{Pfun} \cup \mathit{Pif} \cup \mathit{Pvar} \cup \mathit{Passign} \cup \mathit{Pcall} \cup \mathit{Preturn} \\
\mathit{Pfun} &= \mathsf{Name} \times \text{value-}\mathit{Label}^* \times \text{next-}\mathit{Label} \\
\mathit{Pif} &= \text{condition-}\mathit{Label} \times \text{then-}\mathit{Label} \times \text{next-}\mathit{Label} \\
\mathit{Pvar} &= \mathsf{Name} \times \text{next-}\mathit{Label} \\
\mathit{Passign} &= \mathsf{Name} \times \text{value-}\mathit{Label} \times \text{next-}\mathit{Label} \\
\mathit{Pcall} &= \mathsf{Name} \times \text{value-}\mathit{Label}^* \times \text{ref-}\mathsf{Name}^* \times \text{next-}\mathit{Label} \\
\mathit{Preturn} &= \text{value-}\mathit{Label}
\end{aligned}
$$

In order to define the compilation function, we introduce two auxiliary functions.

$$\mathit{unique} : \mathsf{expression} \rightarrow \mathit{Label}$$

This function is provided by the compiler: a unique labeling of the expressions. Entry labels of expressions are determined by a recursive function

$$\mathit{entryLabel} : \mathsf{expression} \rightarrow \mathit{Label}$$

In the remaining text, we use the following convention: we often write $x.f$ instead of $f(x)$. Function *entryLabel* is recursively defined as follows.

entryLabel(*e* : expression) ≡
  if *e* ∈ sequence then *entryLabel*(*e*.fst-expression)
  elsif *e* ∈ equality then *entryLabel*(*e*.lhs-expression)
  elsif *e* ∈ assignment then *entryLabel*(*e*.expression)
  elsif *e* ∈ ifExpr then *entryLabel*(*e*.test-expression)
  elsif *e* ∈ procReturn ∧ *e*.expression ≠ undef
    then *entryLabel*(*e*.expression)
  elsif *e* ∈ function ∧ *e*.expression-seq ≠ empty
    then *entryLabel*(*e*.expression-seq[1])
  elsif *e* ∈ procCall ∧ *e*.expression-seq ≠ empty
    then *entryLabel*(*e*.expression-seq[1])
  else *unique*(*e*)

Given these domains and functions, we can define the compilation function and the entry labels as below.

6.2 Compilation of Expressions

The compilation of expressions is the main part of the compilation function. We present below a part of it in order to take the idea clear. There is nothing complicated to the omitted parts. We have introduced appropriate comments marked with '//'.

compileExpr(*e* : expression, *next* : *Label*) ≡
  if *e* ∈ variable *// evaluate the variable*
    then {**mk-***Instruction*(*unique*(*e*), **mk-***Pvar*(*e*.Name, *next*))}

```
  elsif e ∈ ifExpr // first evaluate the test, then switch to then or to else
      then compileExpr(e.test-expression, unique(e)) ∪
           { mk-Instruction(unique(e),
             mk-Pif(unique(e.test-expression),
             entryLabel(e.then-expression), elseLabel)} ∪
           compileExpr(e.then-expression, next) ∪
           compileExpr(e.else-expression, next)
       else ...
where
 elseLabel ≡
if e.else-expression = undef then next
                             else entryLabel(e.else-expression)
```

6.3 Compilation of Specifications and Procedures

The compilation of a specification is just the compilation of its main expression. The entry label of a specification is the entry label of its main expression.

```
compileSpec(s : sslSpecification) ≡
  compileExpr(s.main-expression, undef)
entrySpec(p : sslSpecification) ≡
  entryLabel(s.main-expression)
```

The compilation of a procedure is just the compilation of its main expression. The entry label of a procedure is the entry label of its main expression.

```
compileProc(p : procedureDef) ≡
  compileExpr(p.main-expression, undef)∀
p ∈ procedureDef :   entryProc(p.Name) =
                     entryLabel(p.main-expression)
```

The simplified result of the compilation of the sample SSL specification (see figure 2) without procedure hanoi can be found below in figure 3. Apart from the generated code, the following equation hold:
specEntry(`Hanoi`) = *H0*

7 SSLM: An Abstract Machine for SSL

In this section we introduce an abstract machine for SSL called SSLM, corresponding to the SDL Abstract Machine. The intention of our approach is to compile each SSL-instruction to a set of "simple" SSLM instructions, and to give these "simple" instructions a rigorous meaning. Like in [17], and [3], we present the definition of SSLM in several steps. In each step we add some details of a particular aspect of the language SSL. After introducing some basics, we present in the first step the semantics of "simple" primitives like the evaluation of assignments. In a second step we add the meaning of procedure calls and returns. In the last step we add the distribution to SSLM.

Label	Primitive	Next	Comment
H0	fun(4,⟨⟩)	H1	*value of 4*
H1	assign(x,H0)	H2	*x:=4*
H2	fun(0,⟨⟩)	H3	*value of 0*
H3	assign(y,H2)	H4	*y:=0*
H4	fun(0,⟨⟩)	H5	*value of 0*
H5	assign(z,H4)	H6	*z:=0*
H6	fun(4,⟨⟩)	H7	*value of 4*
H7	assign(n,H6)	H8	*n:=4*
H8	var(n)	H9	*value of n*
H9	proc(hanoi,⟨H8⟩,⟨x,y,z⟩)	undef	*hanoi(x,y,z,n)*

Fig. 3. Compilation of Specification Hanoi

7.1 Basics

In this section we describe how states are realized in SSLM. Usually, a state is a mapping of variable names to values. In SSLM locality and scope of variables are given by a model in which variable names are mapped to (memory) locations containing values. This model allows for call-by-value and call-by-reference semantics, i.e. procedures can have reference and value parameters. To this aim we introduce a dynamic domain *Location* for locations and dynamic, controlled functions

$$loc : \mathsf{VarName} \rightarrow Location$$
$$cont : \mathsf{Location} \rightarrow Value.$$

We use here and in the following VarName, FuncName, and ProcName as synonyms for Name. We use in the following a static, controlled function

$$splmInstruction : Label \rightarrow Primitive,$$

which treats the sets of pairs returned by the compilation functions as a function. This is possible since the compilation functions always compute a graph of a function. Furthermore, we introduce a kind of program counter, modeled by

$$currentLabel :\rightarrow Label,$$

which returns the label associated with the instruction to be executed next. During the dynamic execution of a program intermediate values can be stored at labels. More precisely, we introduce the dynamic, controlled function

$$labelValue : Label \rightarrow Value,$$

for this purpose. In case of a sequence $seq = (l_1, \ldots l_n)$ of labels we simply write $labelValues(seq)$ for the sequence $(labelValue(l_1), \ldots, labelValue(l_n))$.
Function names are evaluated by a static function that is provided by the data semantics of the language. This would again be part of the data interface in the case of SDL.

$$evalFunc : \mathsf{FuncName} \times Value^* \rightarrow Value$$

7.2 Simple Primitives

In this section we describe the semantics of simple primitives of SSLM. These primitives are used for the evaluation of expressions, assignments, and conditions. The evaluation of expressions is realized by macros EvalFunc and EvalVar. Constant names and function names are evaluated with the macro EvalFunc. This macro requires three formal parameters: a function name *funcName*, a sequence of labels *labels*, and a continue label *next*. The intention is that values stored at *labels* are extracted with function *valueLabel*. Function *funcName* is evaluated with these values, and the resulting value is stored at *currentLabel*.

EvalFunc$(funcName, labels, next)$ $\equiv$
  $labelValue(currentLabel) := evalFunc(funcName, labelValues(labels))$,
  $currentLabel := next$

Variable names are evaluated with the macro EvalVar in a similar way as function names.This macro requires two formal parameters: a variable name *varName*, and a continue label *next*.

EvalVar$(varName, next)$ $\equiv$
  $labelValue(currentLabel) := cont(loc(varName))$,
  $currentLabel := next$

Assignments are evaluated with the macro EvalAssign. This macro requires three formal parameters: a variable name *varName*, a value label *valueLabel*, and a continue label *next*. Macro EvalAssign essentially sets the value of variable name *varName* to the value stored in *label* and jumps to continue label *next*.

EvalAssign$(varName, label, next)$ $\equiv$
  $cont(loc(varName)) := labelValue(label)$,
  $currentLabel := next$

Furthermore, there exists a macro EvalCondition which evaluates conditions. Due to its simplicity we omit the corresponding definition.

7.3 Procedures

Before we can define macros for procedure call and return we must have some means for storing multiple values of several incarnations of a function. To this aim we introduce frames which are organized as stacks. The universe *Frame* comprises at each point of time a finite set of frames which initially contains one element. A dynamic, controlled function $topFrame :\rightarrow Frame$ indicates the current top of the stack. The function $prev : Frame \rightarrow Frame$ returns for each frame its predecessor (if there is any). The following dynamic, controlled functions are introduced for the procedure call and return mechanism. The former function corresponds to a kind of return address, the latter gives the information at which label the result of a procedure must be stored.

$returnLabel : Frame \rightarrow Label$
$resultLabel : Frame \rightarrow Label$

We change the declaration of function *loc* to

$$loc : \mathsf{Name} \times Frame \rightarrow Value.$$

This means we have to rewrite almost every rule defined so far. Instead of doing so we simply state that every previous occurrence of $f(x)$ should be replaced by $f(x, topFrame)$, where f is one of the functions $\{loc, labelValue\}$ (cf. figure 4). For example, rule EvalVar becomes now

EvalVar$(varName, next)$ $\equiv$
  $labelValue(currentLabel, topFrame) :=$
  $cont(loc(varName, topFrame))$,
  $currentLabel := next$

Macro EvalCall requires four formal parameters: a procedure name *procName*, a sequence of labels *labels*, a sequence of variable names *varNames*, and a return label *label*. A procedure call leads to the creation of a new frame. The new frame is pushed on the stack of frames, and return-label and result-label for this frame are set. New Variable bindings are established by a parameter passing mechanism or by the creation of local variables.

EvalCall$(procName, labels, varNames, label)$ $\equiv$
  extend *Frame* with $frame$
    PushNewFrame$(frame, label)$,
    ParameterPassing$(procName, labels, varNames, frame)$,
    CreateLocalVars$(procName, topFrame)$,
    $currentLabel := procEntry(procName)$

Macro PushNewFrame pushes *frame* on the stack of frames, and updates *returnLabel* and *resultLabel*.

PushNewFrame$(frame, label)$ $\equiv$
  $topFrame := frame, prev(frame) := topFrame,$
  $returnLabel(frame) := label, resultLabel(frame) := currentLabel$

Macro ParameterPassing realizes the parameter passing mechanism, i.e. it handles passing of value and reference parameters.

ParameterPassing$(procName, labels, varNames, frame)$ $\equiv$
  PassRefParameters$(procName, varNames)$,
  PassValParameters$(procName, labels)$,
  StaticBinding$(procName, frame)$

Macro PassRefParameters has two formal arguments: the procedure name *procName* and the sequence of actual reference parameters *varNames*. This macro realizes the parameter passing mechanism for reference parameters. This means references of variable names to locations in current state are changed appropriately. We use a function $refPar : \mathsf{ProcName} \times Nat \rightarrow \mathsf{VarName}$ to determine reference parameters. Function $|.|$ denotes the length function for sequences.

PassRefParameters($procName, varNames$) $\equiv$
  forall $i \in \{1, ..., |varNames|\}$ do
    $loc(refPar(procName, i), topFrame) :=$
    $loc(varNames(i), topFrame)$

Macro PassValParameters creates for each value parameter a new location. Each label in the sequence *labels* has a value in frame. We write simply $labels(i)$ to project on the i-th component of this sequence. The value of a new location *location* is set to the value stored in *labels* and frame where i ranges over $\{1, \ldots, |labels|\}$. The location for the i-th value parameter given by a function $valPar : \mathsf{ProcName} \times Nat \to \mathsf{VarName}$ is set to a new location. We assume that the size of *labels* equals the number of value parameters.

PassValParameters($procName, labels$) $\equiv$
  forall $i \in \{1, ..., |labels|\}$ do
    extend $Location$ with $location$
      $cont(location) := labelValue(labels(i), topFrame)$,
      $loc(valPar(procName, i), topFrame) := location$

Let p be a procedure name. We introduce a static function

$$binding : \mathsf{VarName} \times \mathsf{ProcName} \to \mathsf{ProcName}$$

which returns for each variable name and each procedure name the name of a procedure to which variable name is bound according to the static scope rules. Function $findLocation : \mathsf{VarName} \times \mathsf{ProcName} \times Frame \to Location$ evaluates for a given variable name x the stack starting with *frame* in order to determine the first frame F such that the equation $binding(x, \mathsf{procName}) = proc(F)$ holds. The dynamic, controlled function $proc : Frame \to \mathsf{ProcName}$ returns for each frame the name of the corresponding procedure.

findLocation($varName, procName, frame$) $\equiv$
  if $binding(varName, procName) = proc(frame)$
    then $loc(varName, frame)$
    else $findLocation(varName, procName, prev(frame))$

Macro StaticBinding saves *procName* in *frame* creates a variable binding for those variables which are not defined in the body of *procName*.

StaticBinding($procName, frame$) $\equiv$
  $proc(frame) := procName$,
  forall $varName \in \mathsf{VarName}$ do
    if $binding(varName, procName) \neq procName$
      then $loc(varName, topFrame) :=$
        $findLocation(varName, procName, topFrame)$

Macro CreateLocalVars creates new location for local variables associated with a procedure or a program. Local variables are determined by a means of a static function $locVars : \mathsf{Name} \to \mathsf{Name\text{-}set}$.

CreateLocalVars($name, frame$) $\equiv$
  forall $varName \in locVars(name)$ do
    extend $Location$ with $location$
      $loc(varName, frame) := location$

Macro EvalReturn essentially removes the top frame of the stack, saves the result according to the result label, and jumps to the return label.

EvalReturn($label$) $\equiv$
  $topFrame := prev(topFrame)$,
  $labelValue(resultLabel(topFrame), prev(topFrame)) :=$
    $labelValue(label, topFrame)$,
  $currentLabel := returnLabel(topFrame)$

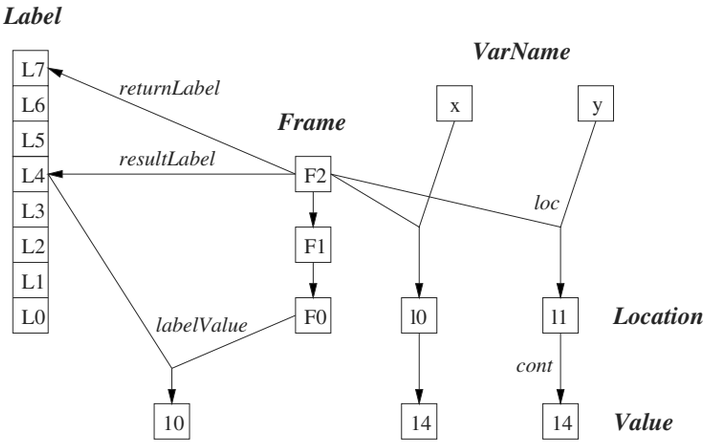


Fig. 4. State and Control Information

7.4 Distribution

In this section we refine the SSLM defined so far to a distributed abstract machine. The underlying distribution model consists of several agents which execute their programs and communicate by means of shared variables. The refinement step is done in a similar way as in the preceding section, namely by changing the declaration of some existing functions. We provide each agent with its own program counter. To this aim we change the declaration of functions *currentLabel*, and *topFrame* to

$currentLabel : Agent \rightarrow Label$,
$topFrame : Agent \rightarrow Frame$.

Every previous occurrence of f inside macros should be replaced by $f(Self)$, where f is one of the functions $\{currentLabel, topFrame\}$.

8 Initialization and Execution

With the abstract machine for SSL being defined, we can now assign a formal semantics to SSL specifications by associating, with each specification, a set of initial states and an ASM program. Please note that conceptually, it is better to think of SSLM programs, because the instructions considered here are SSLM instructions. Formally, the programs below are ASM programs.

8.1 Initialization

In the ASM model, the set of initial states is usually defined by constraints imposed on domains, functions, and predicates. These constraints are required to hold in the first state of each run of the abstract state machine. Initial constraints are stated by formulae that are preceded by the keyword initially, i.e. in a property-oriented style.
As it turned out during work on the formal SDL semantics, a property-oriented definition of the set of initial states, given by the set of SDL agents, SDL agent sets, and link agents together with the initial values of all functions and predicates defined on these agents is rather complex and hard to follow. Therefore, a different approach has been chosen, which may be characterized as a combination of constraints and initial computations: several (simple) constraints determine the so-called *set of pre-initial states*. The actual set of initial states is determined by the initialization phase that starts in a pre-initial state, and creates a set of agents as well as the system structure.
To give a flavor of the approach taken in the formal SDL semantics, we adapt it here to define the set of initial states associated with an SSL specification. A nullary function

$$main :\rightarrow Agent$$

is introduced to refer to the pre-initial system agent. Furthermore, we define

$$topLevelSpec :\rightarrow \mathsf{sslSpec}$$
$$mySpec : Agent \rightarrow \mathsf{sslSpec}$$

The set of pre-initial states is defined by the following set of constraints (strictly speaking, the set of pre-initial states is the set of initial states of the ASM model):

initially $Agent = \{main\}$
initially $sslmInstruction =$
$\bigcup\{p \in \mathsf{sslSpec} : p.compileSpec\} \cup$
$\bigcup\{p \in \mathsf{procedureDef} : p.\mathsf{proc\text{-}Name}.compileProc\}$
initially $main.mySpec = topLevelSpecification$
initially $main.currentLabel = topLevelSpecification.entrySpec$
initially $main.Mod = \mathsf{InitModule}$
initially $Frame = \{fstFrame\}$
initially $main.topFrame = fstFrame$

The value of the static function *sslmInstruction* is determined by the compilation defined in Section 6. It is the complete code of all behavior parts within

the specification. Furthermore, the pre-initial values of certain functions are defined. The function *mySpec* associates with each agent the abstract syntax description of its specification. For agent *main*, this is defined by the nullary function *topLevelSpecification*, that is, the main specification. The start label of a specification is obtained from the function *entrySpec*, as defined in Section 6. Finally, the behavior of agent *main* is determined by assigning the ASM module InitModule, which determines the steps of *main* during the initialization phase:

```
InitModule ≡
  forall p ∈ Self.mySpec.sslSpec-seq do
    extend Agent with a
      extend Frame with frame
        a.mySpec := p, a.topFrame := frame
        a.currentLabel := p.entrySpec, a.Mod := InitModule,
  CreateLocalVars(Self.mySpec.spec-Name, Self.topFrame)
  Self.Mod := ExecModule
```

Firing of InitModule creates, for each SSL specification defined in the specification of the current agent, one agent executing the InitModule. Furthermore local variables of the corresponding SSL specification are created. This leads to a recursive unfolding of the tree of SSL specifications defined by the top level specification. Macro CreateLocalVars is defined in section 7.3.
In the SDL semantics, it is important that the initialization phase is completed before the execution phase starts. This is achieved by a mechanism that blocks agents after their initialization until the initialization phase has ended. For simplicity, we have not modeled this mechanism in the solution presented here.

8.2 Execution of SSL Specifications

The execution of an SSL specification is given by the following ASM module:

```
ExecModule ≡ EvalInstruction(Self.currentLabel.sslmInstruction)
```

The evaluation of an instruction is given by macro EvalInstruction. Depending on the type of an instruction a corresponding macro is executed. For example, an instruction of type *Passign* leads to the execution of macro EvalAssign as illustrated in the following.

```
EvalInstruction(p) ≡
  if ...
  elsif p ∈ Passign
      then EvalAssign(p.Name, pvalue-Label, p.next-Label)
  elsif p ∈ Pcall
      then EvalCall(p.Name, p.value-Label-seq, p.ref-Name-seq, p.next-Label)
  ...
```

9 Conclusions

In this paper, we have presented the transformational/operational approach which has been successfully applied to define the semantics of SDL-2000 formally. We have illustrated some basic techniques of the approach using SSL, a simple specification language.
The following topics have been covered:

- *Abstract Machine:* The semantics is based on an abstract machine. This machine is an extension of the ASM model and is specifically tailored towards the needs of the specification technique. It introduces an abstract code for the FDT and could therefore also be used for its implementation. The abstract machine comprises a set a machine instructions and their semantics as well as some domains and functions. An important part of the abstract machine is the built-in procedure model that allows for call-by-value and call-by-reference parameter passing semantics. Furthermore, the concept of static binding is built into the abstract machine.
- *Compilation:* Any program/specification of the specification technique is given a semantics by transforming it into the abstract machine. This is essentially an abstract compilation function. The compilation transforms the domain of the abstract syntax trees into the domain of abstract machine instruction sets. The execution of these sets is then covered within the description of the abstract machine.
- *Initialization:* In order to run a specification on the abstract machine, a special initialization has to be performed. To this end, the abstract syntax tree is used to determine the internal parts of the specification to be initialized. These parts are then initialized as well. This way, the initialization takes care of structural information, while the behavioral information is handled by the compilation function. The same initialization behavior could also be used for dynamic creation of specifications.

The subdivision of the semantics into the parts presented above has proved very useful in the context of SDL, as it is essential here to be able to adjust the semantics to the development of the FDT. This is achieved by using two layers of abstraction: (1) abstract the concrete syntax into an abstract syntax, (2) transform the abstract syntax to an abstract machine.
It should be noted that the formal SDL semantics has been conceived in parallel to the language definition itself. During this process, several substantial changes to the language definition, which turned out to be a "moving target", were made. These changes of course affected the formal semantics definition, but usually did not touch the SAM.
Finally, it should again be stressed that the work reported here is not an academic exercise, but takes place in a real-life, industrial setting. In our opinion, it is this kind of work academic efforts should eventually lead to. The successful application of mathematical formalism to real-world problems and their approval by industry is a strong selling point for having formalisms at all. In this sense, the work in progress reported in this paper is one important step in this direction.

Acknowledgment. We thank the anonymous referees and the organizers of the ASM2000 workshop for their constructive criticism on a preliminary version of this paper.

References

1. J. A. Bergstra and C. A. Middleburg. Process Algebra Semantics of φSDL. Technical Report UNU/IIST Report No. 68, UNU/IIST, The United Nations University, April 1996.
2. E. Börger. High Level System Design and Analysis using Abstract State Machines". In Traverso Ullman Hutter, Stephan, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, LNCS. Springer, 1999.
3. E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, LNCS. Springer, 1998.
4. M. Broy. Towards a Formal Foundation of the Specification and Description Language SDL. *Formal Aspects of Computing 3*, (3):21–57, 1991.
5. M. Broy, F. Dederichs, C. Dendorfer, M. Fuchs, T. F. Gritzner, and R. Weber. The design of distributed systems – an introduction to FOCUS. Technical Report TUM-19202-2 (SFB-Bericht Nr. 342/2-2/92/A), Institut für Informatik, Technische Universität München, January 1993.
6. J. Ellsberger, D. Hogrefe, and A. Sarma. *SDL: Formal Object-oriented Language for Communicating Systems*. Prentice Hall Europe, 1997.
7. R. Eschbach, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach using Abstract State Machines. In *Local Proc. of ASM2000, Monte Verità, Switzerland*, 2000.
8. J. Fischer and E. Dimitrov. Verification of SDL protocol specifications using extended Petri Nets. In *Proc. of the Workshop on Petri Nets and Protocols of the 16th Intern. Conf. on Application and Theory of Petri Nets*, pages 1–12. Torino, Italy, 1995.
9. J. Fischer, E. Dimitrov, and U. Taubert. Analysis and formal verification of SDL'92 specifications using extended Petri Nets, 1995. internal report.
10. U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 2, pages 271–280, 1998.
11. U. Glässer, R. Gotzhein, and A. Prinz. Towards a New Formal SDL Semantics Based on Abstract State Machines. In *SDL'99 - The Next Millennium, Proc. of the 9th SDL Forum*. Elsevier Science B.V., 1999.
12. U. Glässer and R. Karges. Abstract State Machine semantics of SDL. *Journal of Universal Computer Science*, 3(12):1382–1414, 1997.
13. R. Gotzhein, B. Geppert, F. Rößler, and P. Schaible. Towards a new formal SDL semantics. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 55–64, 1998.
14. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford Univ. Press, 1995.
15. Y. Gurevich. May 1997 draft of the ASM guide. Technical Report CSE-TR-336-97, University of Michigan, 1997.

16. Y. Gurevich. The sequential ASM thesis. *Bulletin of the European Association for Theoretical Computer Science*, 67:93–124, February 1999.
17. Y. Gurevich and J. K. Huggins. The Semantics of the C Programming Language. In *Selected papers from CSL'92 (Computer Science Logic)*, LNCS, pages 274–308. Springer, 1993.
18. U. Hinkel. *Formale, semantische Fundierung und eine darauf abgestützte Verifikationsmethode für SDL.* Ph.D., Techn. Univ. München, 1998.
19. E. Holz and K. Stølen. An Attempt to Embed a Restricted Version of SDL as a Target Language in Focus. In St. Leue D. Hogrefe, editor, *Proc. of Forte '94*, pages 324–339. Chapmann & Hall, 1994.
20. ITU-T. Recommendation Z.100: Specification and Description Language (SDL), International Telecommunication Union (ITU). Geneva, 1999.
21. St. Lau and A. Prinz. BSDL: The Language – Version 0.2. Department of Computer Science, Humboldt University Berlin, August 1995.
22. A. Mitschele-Thiel. Performance evaluation of SDL systems. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. of the 1st Workshop of the SDL Forum Society on SDL and MSC (Berlin, June 29 - July 1, 1998)*, volume 1, pages 35–44, 1998.
23. A. Olsen, O. Færgemand, B. Møller-Pedersen, R. Reed, and J. R. W. Smith. *Systems Engineering Using SDL-92.* Elsevier Science B. V., 1994.
24. A. Prinz, R. Gotzhein, U. Glässer, and R. Eschbach. SDL Formal Semantics, Z.100.F, ITU-T, International Telecommunication Union. `www.informatik.hu-berlin.de/~prinz/Semantics/Z100F_Geneva.doc`.

Description and Simulation of Microprocessor Instruction Sets Using ASMs

Jürgen Teich[1], Philipp W. Kutter[2], and Ralph Weper[1]

[1] Electrical Engineering Department,
University of Paderborn, Germany
{teich, weper}@date.uni-paderborn.de
[2] Department of Electrical Engineering,
Swiss Federal Inst. of Tech. Zurich, Switzerland
philipp@kutter.org

Abstract. In this paper, we describe how cycle-accurate processor behavior may be efficiently described using Abstract State Machines (ASMs). Given a register transfer description of the target processor, an extraction mechanism is described following the approach in [26] that extracts so called *guarded register transfer patterns* from the processor description. It will be shown that these may be directly transformed into a set of ASM rules which in turn provide an executable model of the processor for simulation purposes. Here, we use the ASM description language *XASM* from which the Gem-Mex tool [2] automatically generates a graphical simulator of a given architecture. The feasibility of this approach is demonstrated for an ARM microprocessor.

1 Motivation

Areas such as digital signal processing and process control have brought up new kinds of processor architectures which are dedicated to their specific needs and frequently called ASIPs (application-specific instruction set processors). Examples are digital signal processors (DSPs) and micro-controllers. Especially in the area of embedded systems, efficiency of both the architecture and the compiler for code generation is of utmost importance.

Hence, a new era dealing with the problem of simultaneously developing processor architecture and compiler has just emerged – architecture and compiler co-design.

Here, we describe a method that starts with an extraction method following the approach of Leupers [26] that enables an efficient description and extraction of the instruction set of a given register transfer description of the target architecture. The instruction set of a processor may be described by a set of *guarded register transfer patterns*. Here, these patterns are translated directly into a set of Gurevich's ASM [17] (abstract state machine) rules, a model for parallel computation that has already been widely used for describing the semantics of as well problem-oriented languages (like C [18], Prolog [11], Oberon [22], Java [31]) as hardware description languages (e.g., behavioral VHDL [8]). Several case

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 266–286, 2000.

studies to describe processor architecture hardware (e.g., APE100 [7,6], ARM microprocessor [19]) have successfully demonstrated the capabilities of ASMs for modeling the behavior of hardware and show how those specifications may be refined, e.g., to express pipelining [10,5].

In [12], Del Castillo and Hardt use the specification language ASM-SL for modeling processor behaviors by ASMs. ASM-SL is the basis of a tool environment called "The ASM Workbench" which supports syntax- and type-checking of ASM specifications as well as simulation and debugging of ASMs. Unfortunately, Del Castillo and Hardt do not show how to derive ASM descriptions automatically. So what existing studies of using ASMs for hardware description miss (see [9] for a reference bibliography), is a systematics to automatically derive ASM descriptions from known models, such as register transfer descriptions, e.g., from specifications written in hardware description languages (VHDL [8], MIMOLA [4], nML [14], etc.).

In the following, we list some significant approaches aiming at code generation for retargetable processors.

- The RECORD system [26], developed at the University of Dortmund, aims at automatic code generation for fixed-point DSPs with a fixed instruction word length. The application program, written in a data flow language (DFL), is internally represented by a control-/data flow graph. The target processor has to be specified by the user in the structural hardware description language MIMOLA [4]. RECORD is a further development of the machine-independent MSSQ compiler [28,27] which generates microcode based on a MIMOLA description of the target architecture.
- The hardware description language nML [14] describes a processor in terms of its instruction set. A machine description is organized as an attribute grammar with the derivation rules structuring the description and the attributes defining the instructions' properties.

  For nML, a rather simple model of execution is proposed: a running machine executes a single thread of instructions which are held in a memory and are addressed via a program counter. The nML language permits concise, hierarchical processor description in a behavioral style.
- The CBC/SIGH/SIM environment [13] consists of the retargetable code generator CBC, which uses a standard code-generator generator for instruction selection, and the instruction set simulator SIGH/SIM. Target architectures are described via nML, application programs are given in a high level language. The basic idea of SIGH/SIM is to generate a single C++ function for each instruction of the target processor simulating the impact of the instruction's execution on the processor's state. However, in case of pipelined architectures, this leads to enormous costs because the C++ functions have to be copied for each pipeline stage.
- Another tool using nML is CHESS [16,25], developed at IMEC in Leuven, Belgium, a retargetable code generation environment for fixed-point DSP processors. CHESS uses a mixed behavioral/structural processor model supporting load/store architectures and both homogeneous and heterogeneous

register structures. The application is given as a C- or DFL-program and is represented by a control-/data flow graph. The processor description is represented by a so called instruction set graph containing the register set and a compact description of the instruction set. CHESS supports automatic bit alignment and generates machine code to be simulated by the instruction set simulator CHECKERS [25].

- FLEXWARE (*SGS Thompson, Bell Northern Research*) [29] uses a mixed model for behavioral/structural processor description. Target processors are described by three separate structures: the set of available instruction patterns, a graph model representing the data-path, and a resource classification that accounts for special-purpose registers. The whole framework consists of the retargetable code generator CODESYN and the instruction set simulator INSULIN. CODESYN takes one or more algorithms expressed in a high-level language and maps them onto a user defined instruction set to produce optimized machine code for a target ASIP or a commercial processor core. INSULIN is based on a reconfigurable VHDL model of a generic instruction set processor. Due to the use of standard VHDL tools, the advantage of this approach is not the generation of highly efficient code but the time saving in processor model development [29].
- CASTLE [24] is an automata-theoretic approach describing the behavior of the data-path as *extended finite state machines* which are extracted from a register transfer level description of the target processor based on a VHDL-template.

In this paper, we show how a simulator of a processor architecture given either a netlist or a graphical description of its data-path, may be automatically generated as follows:

1. *Instruction set extraction*: Using ideas of the work of Leupers [26], the instruction set of a processor may be automatically extracted from a processor description resulting in a set of so called *guarded register transfer patterns* that describe the cycle-accurate behavior of the instruction set architecture.
2. *ASM generation*: From this set of patterns, we show that an ASM description that also reflects the cycle-accurate behavior of the processor may be automatically generated. In particular, we present a code translation methodology that generates an ASM description into the language XASM [1].
3. *Simulator/Debugger Generation*: In the last step, the existing ASM simulation and prototyping environment Gem-Mex [23] is used in order to generate a simulator/debugger for this ASM. Major advantage of using this methodology is its efficiency in the number of simulation cycles per second that may be obtained for simulating the execution of a microprocessor.
4. *Generic Simulation Core*: Another highlight of our approach is that the functional behavior of instructions of many different number representations and word-lengths may be considered using an arbitrary precision integer engine.

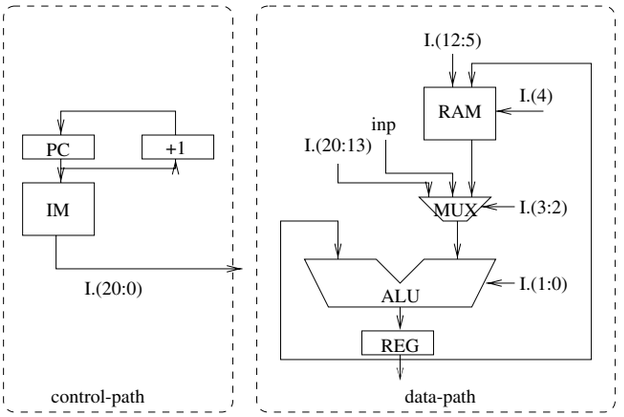


Fig. 1. A simple processor architecture

5. *Library based computing engine*: Based on the above simulation core, a library of generic instructions including arithmetic, transport operations, and number conversions encountered in almost any microprocessor is supported.

2 Generic Description of Instruction Sets by Register Transfer Patterns

During the execution of a machine instruction, several register transfers may take place. In [26], the set of these register transfers is represented by so called *register transfer patterns* (RTP) which are *templates* of register transfers. A *register transfer* (RT) in turn reads input values from registers, memory cells, and/or input ports, performs a computation, and assigns the result to a destination. That way, a RTP reflects the capabilities of the data-path of a microprocessor under question.

Example 1. Fig. 1 shows the architecture of a simple processor consisting of its control- and data-path. *I* stands for instruction word, *inp* denotes an input port, PC and REG denote (scalar) registers. RAM and IM denote memories where IM is the instruction memory. The terms in parentheses (*hi*:*lo*) denote the respective bit index subrange of the instruction word *I*.

Given a processor with a set of registers REG, a set of memories MEM, a set of input ports P_{IN}, and a set of output ports P_{OUT}, we give the following definitions which are close to Leupers' [26]:

Definition 1 (RT Expression). *A register transfer (RT) expression is either*

- *a binary constant* $B \in \{0,1\}^+$,

- *a scalar read access* $readr(r)$ *or* $readp(p)$ *with* $r \in REG, p \in P_{IN}$,
- *an indexed read access* $readm(m, a)$ *where* a *is a register transfer expression and* $m \in MEM$,
- *a complex expression* $op(e_1, \cdots, e_k)$ *where* op *is an operator of arity* $k \geq 1$ *and* $e_1, \ldots, e_k$ *are RT expressions, or*
- *a subrange expression* $e' = e.(hi : lo)$ *where* e *is an RT expression and* $(hi : lo)$ *denotes a bit index subrange.*

Definition 2 (RT Destination). *A register transfer (RT) destination is either*

- *a scalar write access* $writer(r)$ *or* $writep(p)$ *with* $r \in REG, p \in P_{OUT}$, *or*
- *an indexed write access* $writem(m, a)$, *where* a *is a register transfer expression and* $m \in MEM$.

Definition 3 (RT Pattern). *A register transfer pattern (RTP) is a pair* (d, e), *where* d *is a register transfer destination and* e *is a register transfer expression.*

The set of RTPs of a processor under consideration may be extracted automatically by traversing the signal paths from an arbitrary source to each destination d of the target architecture which are executable in one machine cycle.

Example 2. In Fig. 2, the data-path of a simple processor is shown. The RTPs that write $R5 \in REG$ are given by the set:
$\{(writer(R5), +(readr(R1), readr(R3))), (writer(R5), +(readr(R2), readr(R3))),$
$(writer(R5), +(readr(R1), readr(R4))), (writer(R5), +(readr(R2), readr(R4)))\}$.

In general, if register files are the source and target of an instruction, the number of RTPs may be represented by templates, hence, only one RTP is needed for a class of targets, e.g., a register file.

From a hardware-oriented point of view, register transfers take place during the execution of each machine cycle. The execution of a particular register transfer, however, depends on the control-path and the machine state. Therefore, each RTP is coupled with a condition for its execution, a so-called *register transfer condition* (RTC).

Typical conditions for the execution on an RT are

- *static* conditions, e.g., conditions involving condition code, mode registers $MR \subseteq REG$, and instruction bits:
  The mode register variables are defined as the set of Boolean variables

$$MRV = \bigcup_{\mu \in MR} \{M_{\mu,k} \mid M_{\mu,k} \in \{0,1\}; k \in \{0, \cdots, width(\mu) - 1)\}\}$$

  with $width(\mu)$ denoting the width of the mode register μ, the index k denoting the k-*th* bit position of the mode register μ.
  Instruction bit variables are denoted by:

$$IBV = \{I_k \mid I_k \in \{0,1\}; k \in \{0, \cdots, L_w - 1\}\}$$

  with L_w denoting the processor's instruction word length and k indicating a specified bit position.

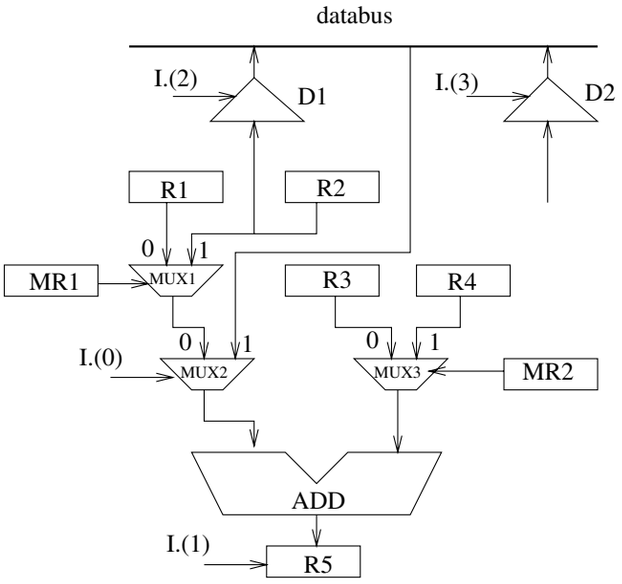


Fig. 2. Simple data-path from which RTPs may be extracted. $I.(0) - I.(3)$ denote corresponding bits of the instruction word I of the control-path

– *dynamic* (run-time) conditions, e.g., the equality of two register contents. Let $COMP$ denote the set of all comparisons permissible on a processor, i.e., the set of RT expressions of the form $op(e, e')$ where $op \in \{=, \neq, <, >, \leq, \geq\}$ is a comparison operator. The dynamic control variables are defined as:

$$DCV = \{D_c \mid D_c \in \{0,1\}; c \in COMP\}$$

Definition 4 (RT Condition). *A* register transfer condition *(RTC) is a Boolean function $F : \{0,1\}^K \rightarrow \{0,1\}$ on the set of Boolean variables $IBV \cup MRV \cup DCV$ with $K = |IBV \cup MRV \cup DCV|$.*

A register transfer is executed if and only if its RTC evaluates to *true* for a given machine state. With the concept of RTCs, we can consider a processor as a machine which in every control step executes the same set of parallel *guarded operations*, each of which has the form

```
IF <register_transfer_condition> THEN <register_transfer_pattern>
```

In general, RTCs may be arbitrary combinations of static and dynamic conditions, and therefore may be considerably complex.

Definition 5 (Guarded Register Transfer Pattern (GRTP)). *A GRTP is a pair (F, T) where F is an RTC and T is an RTP.*

Now, *instruction set extraction* is the task of computing all GRTPs for a given processor. For a computational procedure for instruction set extraction, Leupers uses an internal graph data structure that is generated from a MIMOLA netlist of the processor, see [26] for details. By recursively traversing this graph, the complete set of RTPs may be extracted efficiently.

GRTPs capture all information required for code generation. RTPs represent the RPs that a processor can execute, while RTCs represent the required machine state in terms of partial instructions and mode register states, as well as possible dynamic conditions.

Example 3. Consider again the simple data-path shown in Fig. 2. I stands for instruction word. MR1 and MR2 denote mode registers. Suppose we want to analyze the RTC for RTP "R5:= R2 + R3".[1] Routing the value of R2 to the left input of module ADD via multiplexers MUX1, MUX2 can be realized by control codes $M_{MR1} = 1$ and I.(0) = 0. By setting $M_{MR2} = 0$, the content of R3 is switched to the right input of ADD. Furthermore, loading R5 with the result of the functional unit ADD can be enabled by setting I.(1) = 1. The corresponding RTC is

$$F_1 = \overline{I}.(0) \cdot I.(1) \cdot M_{MR1} \cdot \overline{M}_{MR2}$$

There exists also an alternative route for transportation of R2 to ADD: If bus driver D1 is activated by setting I.(2) = 1, and MUX2 passes its right input (I.(0)=1), then the value of R2 is routed via databus. A potential bus conflict needs to be avoided by setting driver D2 to a tristate mode (I.(3) = 0). The control codes I.(1) = 1 and $M_{MR2} = 0$ remain unchanged. The resulting RTC is

$$F_2 = I.(0) \cdot I.(1) \cdot I.(2) \cdot \overline{I}.(3) \cdot \overline{M}_{MR2}$$

Since F_1 and F_2 are alternative conditions for the same operation, we obtain the GRTP

$$((F_1 \vee F_2), R5 := R2 + R3)$$

2.1 Modeling of Guarded Register Transfer Patterns by ASM Update Rules

In mathematics, domains,[2] functions, and relations constitute what is called a *structure*. Structures without relations are traditionally called *algebras*. A sequential *evolving algebra*, or now called ASM *(abstract state machine)* [17], can be defined by a set of transition rules

```
IF <Cond> THEN <Updates>
```

Note that this formula already looks very much like a GRTP in case `Cond` is an RTC and `Updates` is an RTP. In general, `Cond` may be an arbitrary boolean

[1] Short expression for the RTP $(writer(R5), +(readr(R2), readr(R3))$.
[2] in the following, also called *universes* [17]

valued expression (first-order logic formula) and `Updates` is a finite set of updates to be described later.

The operational semantics of an ASM may be described as follows [17]: The effect of a transition rule $\mathcal{R}$ when applied to an algebra $\mathcal{A}$ is to produce another algebra $\mathcal{A}'$ which differs from $\mathcal{A}$ by the new values for those functions at those arguments where the values are updated by the rule $\mathcal{R}$. If `Cond` is true, the rule can be executed by simultaneously executing each update in the set of updates. In order to understand this, we have to look at the definition of (function) updates because these are the mechanism by which the dynamics of arbitrary systems can be explicitly described.

Definition 6 (Function Update). *Let f denote an arbitrary $n-$ary function and $t_1, \cdots, t_n$ denote a sequence of parameters. Then, a function update*

$$f(t_1, \cdots, t_n) := t$$

sets the value of f at $(t_1, \cdots, t_n)$ to t.

ASMs allow function updates with arbitrary functions f and expressions t_i,t of any complexity or level of abstraction. Functions whose values can change are called *dynamic* in contrast to *static* functions which do not change.

Example 4. Again, we consider the guarded register transfer pattern (GRTP)

$$((F_1 \vee F_2), R5 := R2 + R3)$$

from Example 3 with

$$F_1 = \overline{I}.(0) \cdot I.(1) \cdot M_{MR1} \cdot \overline{M}_{MR2}$$

$$F_2 = I.(0) \cdot I.(1) \cdot I.(2) \cdot \overline{I}.(3) \cdot \overline{M}_{MR2}$$

This pattern may be possibly described by the following guarded function update

$$\texttt{IF } F_1 \vee F_2 \texttt{ THEN } contents(R5) := contents(R2) + contents(R3)$$

in which the set of registers may be modeled by a *universe* REG and the function $contents : REG \rightarrow WORD$ models the RTP for write (read) depending on whether the expression $contents(R_i)$ stands on the left hand side or right hand side of the update. Hence, R5 := R2 + R3 in Example 3 reads the contents of registers $R2$ and $R3$, adds these values, and writes the sum into register $R5$.

In Example 4, we have seen that a guarded register transfer pattern (GRTP) naturally corresponds to a guarded ASM update rule.

In terms of ASM rules, a block of update rules is itself a rule. Hence, in a given state, the updates of all rules whose guards evaluate to true, are executed simultaneously.

Hence, these semantics naturally describe a single-clock hardware implementation: In each clock cycle, all conditions (these are register transfer conditions) are evaluated. Simultaneously, the combinational circuits evaluate the terms they

compute (these are RT expressions). At the end (or beginning) of each clock cycle, the state of the system, as represented by the contents of the registers and memories, is updated. This is represented by guarded updates in which the functions correspond to RT destinations. An ASM description of an instruction set simultaneously defines an interpreter for the instruction set.

3 Generic Processor Simulation - Library and Simulation Core

In this section, it is our purpose to present a set of necessary ASM universe and function definitions that are needed for describing number representations, interconnections, registers and memories, etc. generically.

For functional simulation, we maintain a library of bit-true arithmetic functions that may be used to simulate arbitrary architectures. We define a simulation engine based on arbitrary word-length integer operations such that for different number systems, word-lengths, and word representations (e.g. little versus big endian), only two functions that convert a number into an integer, and, after simulating an operation, back to its number representation, have to be written once.

3.1 Typing and Subtyping

Hardware typically operates on bundles (vectors) of wires, also termed *bit-vectors* or *bit-strings* in hardware description languages such as VHDL. Therefore, it is necessary to define a *simple universe* called $BIT \in \{0, 1\}$ where 0 and 1 denote the numeric values that we usually associate with the Boolean values $false$ and $true$, respectively.

With WORD(n) we denote the universe of all {0,1}-bit-vectors of length n.

$$WORD(n) = \{w \mid w \in \{0,1\}^n; n \in \mathbb{N}\}$$

The same notation is used for registers from the universe REG and memories from the universe MEM in case they have to be distinguished by different word-lengths. For example, let $REG(16)$ denote the subset of registers having a word-length of 16 bits.[3]

For the reason we would like to present processor architectures independent from the number representation used (e.g., signed, one's or two's complement, etc.), we introduce a function[4]

$$val : WORD(t) \rightarrow INTEGER$$

[3] In case all word-lengths are equal for all memories and registers, we will omit the index n for short.

[4] However, we will restrict ourselves in this paper to fixed-point arithmetic.

that returns the value of a word in the corresponding number system and its inverse[5]

$$val^{-1} : INTEGER \rightarrow WORD(t)$$

3.2 Modeling Combinational Modules

A combinational module with n Inputs $i_1, \cdots, i_n$ and m Outputs $o_1, \cdots, o_m$ may be described by a (vector-) function

$$F : i_1 \times \cdots \times i_n \rightarrow o_1 \times \cdots \times o_m$$

Here, $i_1, \cdots, i_n$ are each inputs of a subset of $WORD$, and $o_1, \cdots, o_m$ are each outputs of a subset of $WORD$. In a given number system, the value $val(o_j)$ of each output $o_j, j = 1, \cdots, m$ may be given in terms of the values of the inputs of F by equations of the form:

$$val(o_j) = F_j(val(i_1), val(i_2), \cdots, val(i_n)).$$

3.3 Modeling Registers and Memories

The model for registers (universe REG) and memories (universe MEM) has already been shown. The contents of registers are modeled by unary functions

$$contents : REG(n) \rightarrow WORD(n)$$

for each word-length $n \in INTEGER$. For memories, the indexed access functions

$$contents : MEM(n) \times INDEX \rightarrow WORD(n)$$

are defined for each memory $m \in MEM(n)$ where $INDEX$ is defined over the universe $\{0, \cdots, size(m) - 1\}$ and the function

$$size : MEM(n) \rightarrow INTEGER$$

denoting the size of memory $m \in MEM(n)$ given in terms of the number of words of type $WORD(n)$.

3.4 Modeling Interconnections

It remains to show how to model interconnections, how to extract subwords out of words, how to merge words to larger words, and how to copy bundles of signals.

[5] For non-redundant number systems (e.g., two's complement), the representation of a value $value \in INTEGER$ by a bit-vector of length n is unique. In case the conflict of number representation for the number 0 can be uniquely resolved, also non-redundant number systems such as one's complement and sign-magnitude representations may be dealt with.

Given an element $w \in WORD(n)$ the function *bit* of the form

$$bit : WORD(n) \times INTEGER \rightarrow BIT$$

returns the ith bit b of w — or $b = bit(w, i)$ (or simply $b = w.(i)$) — such that $val(b) = val(bit(w, i))$ in case $0 \leq i < n$, and is undefined else.

In order to compose arbitrary vectors of bits, we introduce a function to model the extraction of subwords from a word, and thus realize the expression $e' = e.(hi : lo)$

$$extract : WORD(n) \times INTEGER \times INTEGER \rightarrow WORD(m)$$

where

$$e' = extract(e, hi, lo)$$

returns a word e' of word-length $m = hi - lo + 1$ in case $lo \leq hi < n$ that satisfies $\forall i \in \{0, hi - lo\} : val(e'.(i)) = val(e.(i + lo))$. Else it is undefined.

Moreover, the function

$$merge : WORD(n) \times WORD(m) \rightarrow WORD(n + m)$$

or $c = merge(a, b)$ allows to merge two words $a \in WORD(n)$ and $b \in WORD(m)$ to form a composed word $c \in WORD(n + m)$ that satisfies $\forall i \in \{0, \cdots, n - 1\} : val(c.(i)) = val(a.(i))$ and $\forall i \in \{n, \cdots, n + m - 1\} : val(c.(i)) = val(b.(i - n))$.

Finally, we define the function

$$split : WORD(n) \rightarrow WORD(n)$$

that allows to copy signals: $b = split(a)$ with $a \in WORD(n)$ and $b \in WORD(n)$ $\forall i \in \{0, \cdots, n - 1\} : val(b.(i)) = val(a.(i))$.

Example 5. Lets consider Fig. 3 where a bundle of wires is grouped from a signals $b \in WORD(6)$, $a \in WORD(8)$, and 5 leading zeros into a 16 bit signal $c \in WORD(16)$. Using *merge* and *extract*, this grouping may be functionally described as follows:

$$c = merge(merge(extract(b, 5, 3), extract(a, 0, 0)), merge(extract(a, 7, 1), zero))$$

3.5 Retargetability to Multiple Number Systems

In the following, we assume that $BIT = WORD(1) \in \{0, 1\}$ is modeling the Boolean values of $false$ and $true$, respectively.

A bit-vector $x \in WORD(n)$ represents a binary vector of length n. This is an ordered sequence of the form $(x_{n-1}x_{n-2} \cdots x_1x_0)$ where the $x_i = bit(x, i)$ are the bits (also called digits). We have introduced the function *val* returning the value of a bit-string. Intentionally, this has been done independently from the *number system* used, e.g., *non-redundant, weighted*, and *positional* number systems. Our intention is to simulate processor architectures bit-true at a high

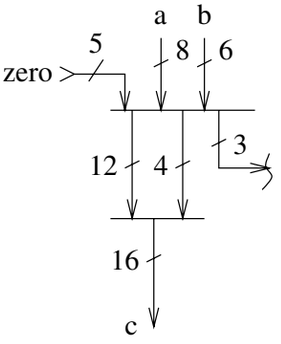


Fig. 3. Example of extract and merge of signals

level (i.e., RT-level) of abstraction, fast and independent from the number system of the architectures' arithmetic-logical units.

Therefore, we implement a simulation engine that computes functions based on (arbitrary long) integer representations for implementation of highly complex arithmetic functions, and use the two functions

$$val : WORD(n) \rightarrow INTEGER \qquad \text{and} \qquad val^{-1} : INTEGER \rightarrow WORD(n)$$

to convert numbers from different number systems. Only these two functions have to be implemented to retarget the simulator to a different arithmetic number system, see Fig. 4.[6] The functions val, respectively val^{-1} are the interfaces of the integer engine to a processor with a different number representation and word-length. Note that in some cases, it may be more convenient to perform operations on the values of extracted subwords (e.g. single bits) of operands using *extract* and not on the values of the complete operands.

In [30], details are given for the definition of the functions val and val^{-1} for typical number systems used in computer systems.

In the following example, it is shown how a typical operation of two operands is simulated generically using the above retargetable simulation engine.

Example 6. Let $c = op(a, b)$ with $a \in WORD(n), b \in WORD(m)$, and $c \in WORD(q)$ denote an arbitrary operation with two operands a and b. The function op is evaluated as follows:

1. compute $val(a)$ based on its word-length n;
2. compute $val(b)$ based on its word-length m;
3. calculate $val(c')$ in terms of $val(a)$ and $val(b)$ according to op;

[6] Note that the view shown is completely opposite to the view one has when simulating hardware. The user (environment) thinks in terms of values (e.g., integers) whereas internally, the hardware realizes operations on bit-vectors in a binary number system.

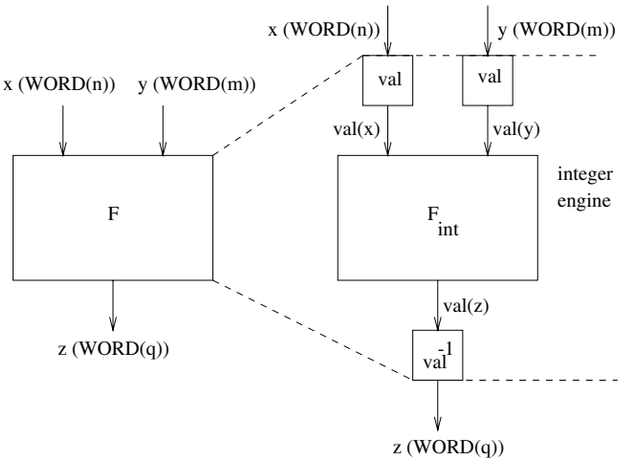


Fig. 4. Concept of a retargetable simulation engine

4. let $val(c') := val(c') \bmod 2^q$ (truncation to result word-length default);[7]
5. let $c = val^{-1}(val(c'))$.

In case $n \neq m$, the above extension of the number representation might be necessary in hardware, however not in our simulation engine as the function *op* is generic. Such an operation is typical for arithmetic computation.

In [30], we present a catalogue of standard computations (e.g. logical and standard arithmetic operations) that have been implemented in the library.

4 An Environment for Processor Simulation and Debugging Using XASM

In this section, we show how the exemplary description of the ARM [15] microprocessor is derived using the above methodology, how the instruction set may be simulated, and how assembler programs of this processor may be debugged.

The motivation of the following ARM-processor case study is a hand written ASM-description as given in [19,20]. In our methodology, a corresponding ASM-description will be obtained by instruction set extraction and ASM generation from a netlist or graphical description of the ARM's data-path.

In Section 4.1 we sketch the architecture of the ARM processor and show how its description is implemented using the ASM language XASM.

In addition to entering the ASM rules, we had to implement the library of arithmetic and logical functions described in Section 3. These functions are implemented in C and interfaced to XASM. Further, a parser both for assembler

[7] Different other behaviors like rounding etc. may be handled similarly.

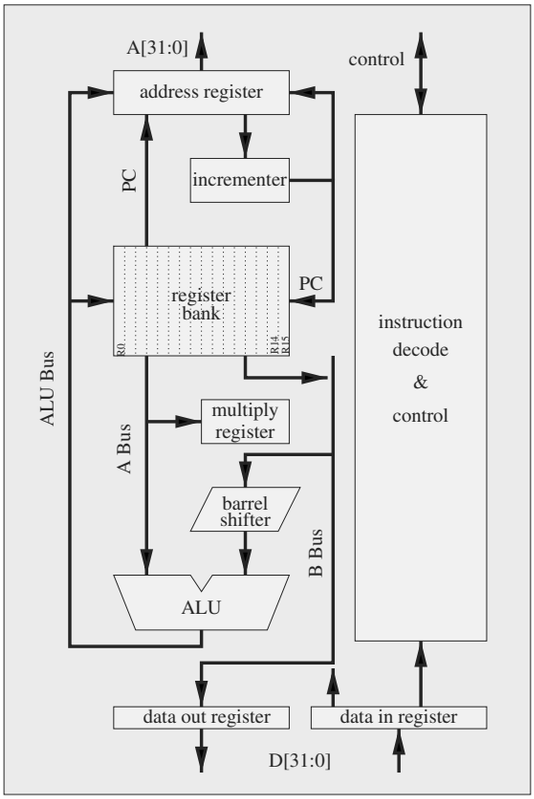


Fig. 5. Architecture of the ARM processor

code and binary code was written using the existing parser support of XASM. Some details are given in Section 4.2. Given the XASM specification, the GemMex tool [2] generates a graphical debugger and simulation tool, see Section 4.3.

4.1 The ARM2 Processor

The ARM2 Processor of *Advanced Risc Machines* [15] is a 32 bit microprocessor architecture with a set of 16 registers, see Figure 5. The dedicated register R15 contains the program counter (PC) and status flags, R14 is reserved for the return address of branch instructions. The registers R0 to R13 are general purpose registers.

The ARM processor supports a three staged pipeline. Instructions are dispatched in three phases: During the first phase *instruction fetch*, an instruction is fetched from the instruction memory, in the second phase *decode*, the instruc-

tion is decoded and assigned to the respective execution unit. During the phase *execute*, the instruction will be executed and the result will be stored.

One can think of these phases as being executed sequentially, but in fact the architecture has parallel logic for all three phases, so in a stable pipeline state, the ARM processor is able to execute three operations of a machine program in parallel. In [19], a simple ASM-model of the ARM microprocessor for sequential execution mode is given and stepwise refined to a model executing the stages in parallel. The correctness of the refinements is proven.

We started by entering the specifications of the different refinements, and debugged them. The disadvantage of considering several refinements simultaneously is that it is almost impossible to maintain all versions if changes must be made. Thus, we decided to integrate the different models into one model that can be transformed into any of the refinements by exchanging some macro definitions. This version of the description is given in [21]. Depending on the boolean value of the macro `PIPELINED`, either the stages are executed sequentially or in pipelined mode. In sequential mode, a nullary function

```
function  Stage
```

is used. The values of `Stage` are fetch, decode, or execute. In XASM syntax, the three constants can be introduced by enumeration:

```
universe STAGES = {fetch, decode, execute}
```

The control structure simulating both the sequential and the parallel execution of stages is structured as follows. Assume, `code[Contents(PC)]` fetches the current instruction from memory.

```
if PIPELINED or Stage = fetch then
   Instr := code[Contents(PC)]
   if PIPELINED then
     DecodeInstr := Instr
   else
     Stage := decode
   endif
   R_FETCH
endif
if PIPELINED or Stage = decode then
   if PIPELINED then
     ExecuteInstr := DecodeInstr
   else
     Stage := execute
   endif
   R_DECODE
endif
if PIPELINED or Stage = execute then
   if PIPELINED then
   else
```

```
      Stage := fetch
    endif
    R_EXECUTE
  endif
```

In pipelined mode, all three parts of the rule are executed in parallel, otherwise the parts are executed sequentially in cyclic mode.

In the three remaining rules R_FETCH, R_DECODE, and R_EXECUTE of the complete ARM description not given here, the active instructions must be accessed by three macros defined as follows:

```
#define FETCHINSTR   Instr
#define DECODEINSTR  (if PIPELINED then DecodeInstr else Instr)
#define EXECUTEINSTR (if PIPELINED then ExecuteInstr else Instr)
```

If these macros are used to refer to the instruction, almost the complete functionality of the rules R_FETCH, R_DECODE, and R_EXECUTE can be defined independent of whether the stages are executed in parallel or sequentially.

In order to make the description retargetable to other architectures, we used the ASM-library of Section 3. The adaption to our methodology involved mainly the adaption of the signature. The structure of the rules remained unchanged.

4.2 Parser Generation for Assembler and Binary Code

XASM provides a number of built in constructs for generating a parser. The main structure of the assembler parser starts with the declaration of a list of instructions, separated by semicolons.

```
nonterm ARMcode[Instruction];
  base Instruction
  cont ARMcode ‘‘;’’ Instruction
endnonterm
```

Instructions are alternatively alu-, branch-, single transfer-, or multiple transfer instructions. In XASM this alternative is written as

```
nonterm Instruction = AluInstr | BranchInstr |
                      SingleTransferInstr | MultipleTransferInstr
endnonterm
```

Finally, the syntax of single instructions is given, for instance for a branch:

```
nonterm BranchInstr ::= ‘‘B’’ (link=)b_link (cond=)Conds
                        (adr=)address;
R
endnonterm
```

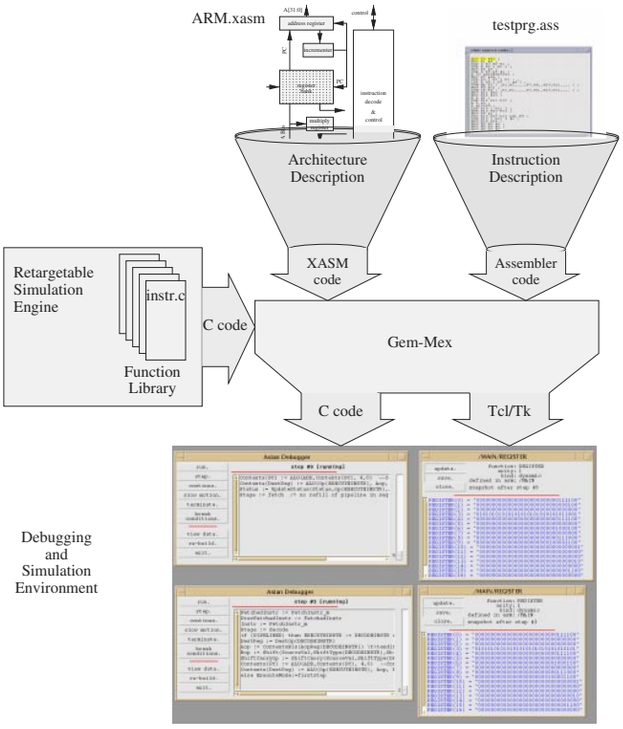


Fig. 6. Designflow from architecture description and program input to simulation

where in the rule `R` one can access the components of the branch instruction as `link`, `cond`, and `adr`. In the rule `R` certain informations about the instruction are precomputed.

If a further analysis of the code is desired, e.g. an abstract verification of certain properties, the Montages [3] support of Gem-Mex can be used. We restricted the use of Gem-Mex to the generation of graphics for simulation and debugging.

4.3 Simulation of the ARM Processor

In Figure 6, we display the designflow from architecture description and program input to simulation. From a graphical description of the architecture, the GRTP description is extracted and then transformed into ASM rules. The upper left part shows the transformation from the architecture description into XASM code resulting in a file ARM.xasm. The upper right part displays the input of the simulator's algorithmic load which consists of an application program written in assembler notation, the file testprg.ass. The XASM-compiler translates

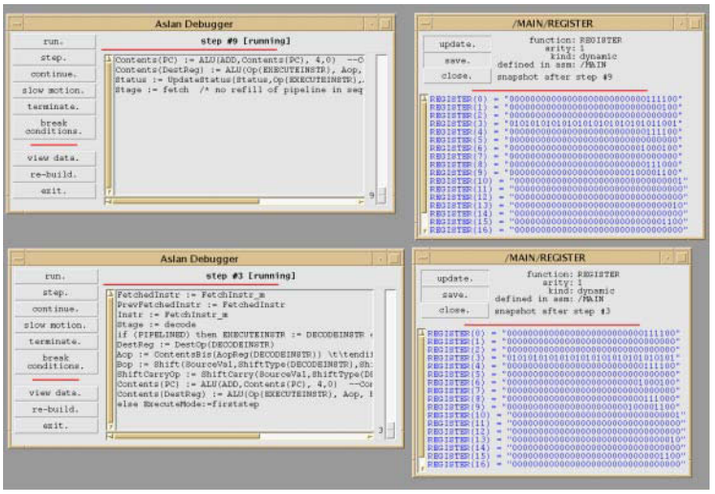


Fig. 7. Debugging of a program using the Gem-Mex Environment

the architecture description into C-code and calls the gcc-compiler to compile these C-files. Thereby, the C-library of arithmetic functions described in Section 3 is linked. As a result, an executable is generated and which interacts with generic Tcl/Tk scripts provided by the Gem-Mex simulation and debugging environment. The visual aspects of the simulation and debugging environment are shown at the bottom of the figure.

Figure 7 displays a snapshot of a simulation run both of the pipelined (top) and the sequential (bottom) implementation. The algorithmic load respectively consists of a small assembler program, see Figure 8.

Here, after nine steps, the sequential implementation has reached the end of an execution phase and sets the `Stage` flag to `fetch` indicating that the next operation has to be a fetch operation. The parallel implementation only needs three cycles to reach the same machine state due to pipelining. The right hand picture shows the contents of the respective register file.

5 Conclusions

In this paper, we present foundations for the simulation of a processor architecture and its instruction set given the syntax of the instruction and either a netlist or graphical description of the architecture's data-path.

A major advantage of our approach with respect to other approaches is that the ASM-formalism itself is suited for architecture description. The generated ASMs can thus serve as additional documentation. Changes may be made directly at the ASM level and the corresponding changes in the netlist/graphical description can be regenerated.

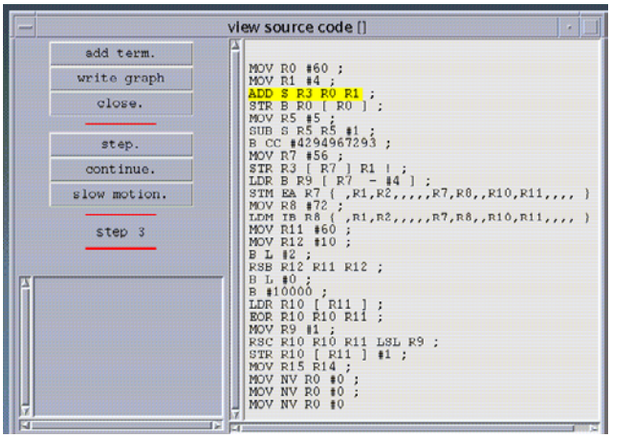


Fig. 8. The application program in ARM assembler language

Furthermore, we present a methodology for generic processor simulation that is independent of a processor's specific parameters like instruction word length and number representation. In order to accomplish this, we build a library of bit-true arithmetic functions and construct an integer engine as a simulation core which can be adapted to almost any microprocessor by specific interface functions.

Here, we demonstrate the feasibility of our methodology for simulating a realistic pipelined RISC architecture, the ARM microprocessor.

References

1. M. Anlauff. Xasm - an extensible, component-based abstract state machines language. In *This Volume*.
2. M. Anlauff, P.W. Kutter, and A. Pierantonio. Formal aspects of and development environments for Montages. In *Second Int. Workshop on the Theory and Practice of Algebraic Specifications*, Workshops in Computing. Springer-Verlag, 1997.
3. M. Anlauff, P.W. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In *Perspective of System Informatics*, number 1726 in LNCS, 1999.
4. S. Bashford, U. Bieker, B. Harking, R. Leupers, P. Marwedel, A. Neumann, and D. Voggenauer. The MIMOLA language - version 4.1. Technical report, University of Dortmund, 1994.
5. E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Springer Lecture Notes on Computer Science (LNCS)*, pages 236–271, 1995.

6. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (the APE100 reverse engineering study). In B. Werner, editor, *Proc. of the First Int. Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145–148, November 1995.
7. E. Börger, G. Del Castillo, P. Glavan, and D. Rosenzweig. Towards a mathematical specification of the APE100 architecture: the APESE model. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I : Technology/Foundations, pages 396–401, Elsevier, Amsterdam, The Netherlands, 1994.
8. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *European Design Automation Conference (EURO-DAC'94)*, pages 500–505, IEEE CS Press, Los Alamitos, CA, U.S.A., 1994.
9. E. Börger and J. K. Huggins. Abstract state machines 1988-1998: Commented ASM bibliography. In H. Ehrig, editor, *Formal Specification Column, Bulletin of the EATCS64*. February 1998.
10. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Springer Lecture Notes on Computer Science (LNCS)*, pages 151–187, 1996.
11. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995. North-Holland.
12. G. Del Castillo and W. Hardt. Fast dynamic analysis of complex hw/sw-systems based on abstract state machine models. In *Proc. of the 6th Int. Workshop on Hardware/Software Co-Design (Code/Cashe98), Seattle*, pages 77–81, March 1998.
13. A. Fauth. Beyond tool-specific machine descriptions. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 138–152. Kluwer Academic Press, 1995.
14. A. Fauth, J. Van Praet, and M. Freericks. Describing instruction set processors using nML. In *Proceedings on the European Design and Test Conference, Paris, France*, pages 503–507, March 1995.
15. S . Furber. *ARM System Architecture*. Addison-Wesley Longman, 1996.
16. G. Goossens, J. Van Praet, D. Lanneer, W. Geurts, and F. Thoen. Programmable chips in consumer electronics and telecommunications. In G. de Micheli and M. Sami, editors, *Hardware/Software Co-Design*, volume 310 of *NATO ASI Series E: Applied Sciences*, pages 135–164. Kluwer Academic Press, 1996.
17. Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36, Oxford University Press, 1995.
18. Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Springer Lecture Notes on Computer Science (LNCS)*, pages 274–309, 1993.
19. J. Huggins and D. Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. Technical report, CSE-TR-321-96, EECS Dept., Univ. of Michigan, 1996.
20. J. Huggins and D. Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Transactions on Design Automation of Electronic Systems*, 3(4):563–580, 1998.
21. S. Küng. Simulation mit Abstrakten Zustandsmaschinen, Studienarbeit, 1998.
22. P.W. Kutter and A. Pierantonio. The formal specification of Oberon. *Journal of Universal Computer Science*, 3(5):443–503, 1997.
23. P.W. Kutter and A. Pierantonio. Montages: Specification of realistic programming languages. *Journal of Universal Computer Science*, 3(5):416–442, 1997.

24. M. Langevin, E. Cerny, J. Wilberg, and H.-T. Vierhaus. Local microcode generation in system design. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 171–187. Kluwer Academic Press, 1995.
25. D. Lanneer, J. Van Praet, A. Kifli, K. Schoofs, W. Geurts, F. Thoen, and G. Goossens. Chess: Retargetable code generation for embedded dsp processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 85–102. Kluwer Academic Press, 1995.
26. R . Leupers. *Retargetable Code Generation for Digital Signal Processors*. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997.
27. R. Leupers and P. Marwedel. Retargetable code compilation based on structural processor descriptions. *Design Automation for Embedded Systems*, 3(1):1–36, January 1998.
28. L. Nowak. Graph based retargetable microcode compilation in the MIMOLA design system. In *20th Annual Workshop on Microprogramming (MICRO-20)*, pages 126–132, 1987.
29. P. Paulin, C. Liem, T. May, and S. Sutarwala. Flexware: A flexible firmware development environment for embedded systems. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*, pages 67–84. Kluwer Academic Press, 1995.
30. J. Teich, P.W. Kutter, and R. Weper. A retargatable engine for the simulation of microprocessors using asms; DATE-report 04/00. 2000.
31. C. Wallace. The Semantics of the Java Programming Language: Preliminary Version. Technical Report CSE-TR-355-97, EECS Dept., University of Michigan, December 1997.

Symbolic Analysis of Transition Systems*

Natarajan Shankar

Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA
{shankar, owre}@csl.sri.com
URL: http://www.csl.sri.com/~shankar/
Phone: +1 (650) 859-5272 Fax: +1 (650) 859-2844

Abstract. We give a brief overview of the *Symbolic Analysis Laboratory* (SAL) project. SAL is a verification framework that is directed at analyzing properties of transition systems by combining tools for program analysis, model checking, and theorem proving. SAL is built around a small intermediate language that serves as a semantic representation for transition systems that can be used to drive the various analysis tools.

The transition system model of a program consists of a state type, an initialization predicate on this state type, and a binary next-state relation. The execution of a program starts in a state satisfying the initialization predicate so that each state and its successor state satisfy the next-state relation. Transition systems are a simple low-level model that have none of the semantic complications of high-level programming languages. Constructs such as branches, loops, and procedure calls can be modelled within a transition system through the use of explicit control variables. The transition system model forms the basis of several formalisms for several popular formalisms including UNITY [15], TLA [28], SPL [31], and ASMs [20]. It also underlies verification tools such as SMV [30], Murphi [18], and STeP [32].

If we focus our attention on the verification of properties of transition systems, we find that even this simple model poses some serious challenges. The verification of transition systems is performed by showing that the system satisfies an invariance or progress property, or that it refines another transition system. It is easy to write out proof rules for the verification of such properties but the actual application of these proof rules requires considerable human ingenuity. For example, the verification of invariance properties requires that the invariant be inductive, i.e., preserved by each transition. A valid invariant might need to be strengthened before it can be shown to be inductive. Fairness constraints and progress measures have to be employed for demonstrating progress

* This work was funded by the Defence Advanced Research Projects Agency under Contract NO. F30603-96-C-0204, and NSF Grants No. CCR-9712383 and CCR-9509931. The SAL project is a combined effort between SRI International, Stanford University, and the University of California, Berkeley.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 287–302, 2000.

properties. It takes a fair amount of effort and ingenuity to come up with suitable invariant strengthenings and progress measures.

Methods like model checking [13] that are based on state-space exploration have the advantage that they are largely automatic and seldom require the fine-grain interaction seen with deductive methods. Since these methods typically explore the reachable state space (i.e., the strongest invariant), there is no need for invariant strengthening. Progress measures are also irrelevant since the size of the whole state space is bounded. However, model checking methods apply only to a limited class of systems that possess small, essentially finite state spaces.

Theorem proving or model checking are not by themselves adequate for effective verification. It is necessary to combine the expressiveness of the deductive methods with the automation given by model checking. This way, small, finite-state systems can be directly verified using model checking. For larger, possibly infinite-state systems, theorem proving can be used to construct *property-preserving abstractions* over a smaller state space. Such abstractions convert data-specific characteristics of a computation into control-specific ones. The finite-state model constructed by means of abstraction can be analyzed using model checking. It is easy to actually compute the properties of a system from a finite-state approximation and map these properties back to the original system.

We give an overview of an ongoing effort aimed at constructing a general framework for the integration of theorem proving, model checking, and program analysis. We use the term *symbolic analysis* to refer to the integration of these analysis techniques since they all employ representations based on symbolic logic to carry out a symbolic interpretation of program behavior. The framework also emphasizes *analysis*, i.e., the extraction of a large number of useful properties, over *correctness* which is the demonstration of a small number of important properties. The framework is called the *Symbolic Analysis Laboratory* (SAL). We motivate the need for symbolic analysis and describe the architecture and intermediate language of SAL.

1 A Motivating Example

We use a very simple and artificial example to illustrate how symbolic analysis can bring about a synergistic combination of theorem proving, model checking, and program analysis. The example consists of a transition system with a state contain a (control) variable PC ranging over the scalar type $\{inc, dec\}$, and two integer variables B and C. Initially, control is in state inc and the variables B and C are set to zero. There are three transition rules shown below as guarded commands:

1. When $PC = inc$, then B is incremented by two, C is set to zero, and control is transferred to state dec.

$$PC = inc \longrightarrow B' = B + 2; C' = 0; PC' = dec;$$

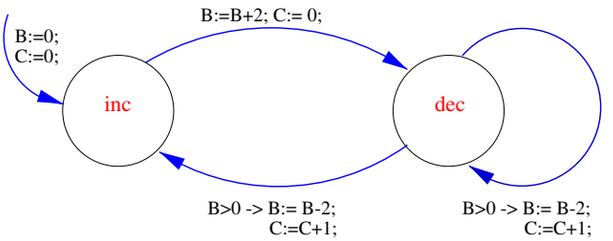


Fig. 1. A Simple Transition System: Twos

2. When $PC = dec$, B is decremented by two, and C is incremented by one, and control is transferred to state *decl*.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1; PC' = inc;$$

3. Same as transition rule 2, but control stays in *dec*.

$$PC = dec \wedge B > 0 \longrightarrow B' = B - 2; C' = C + 1;$$

There is also an implicit stuttering transition from state *dec* to itself when none of the guards of the other transitions holds, i.e., when $B \leq 0$. Since the *inc* state has a transition with a guard that is always true, there is no need for a stuttering transition on *inc*. The transition system is shown diagrammatically in Figure 1.

The transition system *Twos* satisfies a number of interesting invariants.

1. B is always an even number.
2. B and C are always non-negative.
3. B is always either 0 or 2.
4. B is always 0 in state *inc*.
5. C is always either 0 or 1.
6. In state *dec*, $B = 2$ iff $C = 0$.

The purpose of symbolic analysis is to find and validate such properties with a high degree of automation and minimal human guidance and intervention. While efficient automation is essential for analyzing large transition systems, the intended outcome of symbolic analysis is human insight. The analysis should therefore not rule out human interaction.

2 Some Symbolic Analysis Techniques

We enumerate some symbolic analysis techniques and assess their utility on the *Twos* example. For this purpose, we focus on the invariant (1) below.

$$B = 0 \vee B = 2 \tag{1}$$

Note that the transition system *Twos* is a potentially infinite state system since variables B and C range over the integers.

Some mathematical preliminaries are in order. A transition system P is given by a pair $\langle I_P, N_P \rangle$ consisting of an initialization predicate on states I_P, and a binary next-state relation on states N_P. We constrain the next-state relation N to be total so that $\forall s : \exists s' : N(s, s')$. The metavariables s, s' range over states. We treat a set of states as equivalent to its characteristic predicate. The boolean connectives $\wedge$, $\vee$, $\supset$, are lifted from the booleans to the level of predicates and correspond to the set-theoretic operations $\cap$, $\cup$, and $\subseteq$, respectively. An assertion is a predicate on states. The metavariables ϕ, ψ range over assertions. A predicate transformer is a map from predicates to predicates. A monotone predicate transformer τ preserves the subset or implication ordering on predicates so that if $\phi \supset \psi$, then $\tau(\phi) \supset \tau(\psi)$. The fixed point of a monotone predicate transformer τ is an assertion ϕ such that $\phi \equiv \tau(\phi)$. As a consequence of the Tarski–Knaster theorem, every monotone predicate transformer has a least fixed point $lfp(\tau)$ and a greatest fixed point $gfp(\tau)$ such that

$$lfp(\tau) = \tau(lfp(\tau)) \supset gfp(\tau) = \tau(gfp(\tau)).$$

Let $\bot$ represent the empty set of states, $\top$ the set of all states, and ω the set of natural numbers. If the state space is finite, then the least fixed point $lfp(\tau)$ can be calculated as

$$\bot \vee \tau(\bot) \vee \tau^2(\bot) \vee \ldots \vee \tau^n(\bot)$$

for some n, and similarly, $gfp(\tau)$ can be calculated as

$$\top \wedge \tau(\top) \wedge \tau^2(\top) \wedge \ldots \wedge \tau^n(\top),$$

for some n.

If τ is $\vee$-continuous (i.e., $\tau(\bigvee_{i\in\omega} \phi_i) = \bigvee_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_i \supset \phi_j$), then

$$lfp(\tau) = \bigvee_{i\in\omega} \tau^i(\bot) \tag{2}$$

Similarly, if τ is $\wedge$-continuous (i.e., $\tau(\bigwedge_{i\in\omega} \phi_i) = \bigwedge_{i\in\omega} \tau(\phi_i)$ for ϕ_i such that whenever $i < j$, $\phi_j \supset \phi_i$), then

$$gfp(\tau) = \bigwedge_{i\in\omega} \tau^i(\top) \tag{3}$$

Equations (2) and (3) provide an iterative way of computing the least and greatest fixed points but these on infinite-state spaces, the computations might not converge in a bounded number of steps.

Typical examples of monotone predicate transformers include

1. Strongest postcondition of a transition relation N, $sp(N)$, which is defined as
$$sp(N)(\phi) \equiv (\exists s' : \phi(s') \wedge N(s', s)).$$

2. Strongest postcondition of a transition system P, $sp(P)$ is defined as

$$sp(P)(\phi) \equiv I_P \vee sp(N_P)(\phi).$$

3. Weakest precondition of a transition relation N, $wp(N)$ is defined as

$$wp(N)(\phi) \equiv (\forall s' : N(s, s') \supset \phi(s)).$$

2.1 Invariant Proving

The invariance rule is the most heavily used proof rule in any program logic [23, 33]. Given a transition system P as a pair $\langle I_P, N_P \rangle$, consisting of an initialization I_P and a next-state relation N_P, the invariance rule usually has the form:

$$\frac{\begin{array}{c} \vdash \psi(s) \supset \phi(s) \\ \vdash I_P(s) \supset \psi(s) \\ \vdash \psi(s_0) \wedge N_P(s_0, s_1) \supset \psi(s_1) \end{array}}{P \models \textbf{invariant } \phi}$$

In this rule, the assertion ψ is a strengthening of the assertion ϕ. Such a strengthening is needed since the assertion ϕ may not be inductive, i.e., satisfy the premises $\vdash I_P(s) \supset \phi(s)$ and $\vdash \phi(s_0) \wedge N_P(s_0, s_1) \supset \phi(s_1)$.

In the *Twos* example, the invariant (1) is not inductive. It fails because it is not preserved by transition 1 since we cannot establish

$$\begin{array}{ll} \vdash & (PC = inc \wedge (B = 0 \vee B = 2)) \\ & \wedge\ (PC = inc \wedge B' = B + 2 \wedge C' = 0 \wedge PC' = dec) \\ & \supset (B' = 0 \vee B' = 2). \end{array}$$

The invariant has to be strengthened with the observation that when $PC = inc$, B is always 0 so that it now reads

$$B = 0 \vee (PC \neq inc \wedge B = 2). \tag{4}$$

The strengthened invariant (4) is inductive. The need for invariant strengthening in program proofs is the key disadvantage of the deductive methods with respect to model checking. Quite a lot of effort is needed to turn a putative invariant into an inductive one. Once an invariant has been strengthened in this manner, it can contain a large number of conjuncts that generate a case explosion in the proof. Much of the focus of symbolic analysis is on supplementing deductive verification with the means of automatically obtaining useful invariants and invariant strengthenings.

2.2 Enumerative Model Checking

The early approaches to model checking were based on the feasibility of computing fixed point properties for finite-state systems. The reachable states of a

finite-states can be computed by starting from the set of initial states and exploring the states reachable in n consecutive transitions. Any property that holds on all the reachable states is a valid invariant. There are many variations on this basic theme. Many modern enumerative model checkers such as Murphi [18] and SPIN [24] carry out a depth-first search exploration of the transition graph while maintaing a hash-table to record states that have already been visited. In SPIN, the LTL model checking problem is transformed into one of emptiness for ω-automata, i.e., automata that recognize infinite strings [38,19].

In enumerative model checking, properties written in a branching-time temporal logic CTL can be verified in time proportional to $N \times F$ where N is the size of the transition graph and F the size of the temporal formula. Model checking linear-time temporal logic formulas is more expensive and takes time proportional to $N \times 2^F$ where N is the size of the model and F is of the formula.

The *Twos* example succumbs rather fortuitously to enumerative model checking. Even though the potential state space of *Twos* is unbounded, only a bounded part of the state space is reachable since B is either 0 or 2, and C is either 0 or 1. The success of enumerative model checking is somewhat anomalous since this method is unlikely to terminate on typical infinite-state systems. Even on finite-state systems, an enumerative check is unlikely to succeed because the size of the searchable state space can be exponential in the size of the program state. Still, enumerative model checking is an effective debugging technique that can often detect and display simple counterexamples when a property fails.

2.3 Symbolic Model Checking

The use of symbolic representation for the state sets was proposed in order to combat the state explosion problem in enumerative model checking [4,30]. A symbolic representation for boolean functions based on binary decision diagrams (BDDs) [10] has proved particularly successful. A finite state can be represented as a bit-vector. Then sets of bit-vectors are just boolean functions and can be represented as BDDs. In particular, the initial set, a given invariant claim, the transition relation, and the reachable state set, can all be represented as BDDs. The BDD operations can be used to compute images of state sets with respect to the transition relation. This allows predicate transformers such as strongest postcondition and weakest precondition to be applied to the BDD representation of a state set. The reachable state set can be computed by means of a fixed point iteration of the strongest postcondition computation starting from the initial state set. Every intermediate iteration of the reachable state set is also represented as a BDD. There are several advantages to the use of BDDs. Sometimes even sets of large cardinality might have compact symbolic representations. BDDs are a canonical representation for boolean functions so that equivalence tests are cheap. BDDs are especially good at handling the boolean quantification that is needed in the image computations. Automata-theoretic methods can also be represented in symbolic form. Some symbolic model checkers include SMV [30]

Such symbolic representations do require the state to be explicitly finite. This means that the *Twos* example cannot be coded directly in a form that can be directly understood by a symbolic model checker. Some work has to be done in order to reduce the problem to finite-state form so that it can be handled by a symbolic model checker.

2.4 Invariant Generation

Automatic invariant generation techniques have been studied since the 1970s [14, 21,27,36], and more recently in the work of Bjørner, Browne, and Manna [3], and Bensalem, Lakhnech, and Saïdi [9,34,7].

As in model checking, the basic operation in invariant generation is that of taking the strongest postcondition or weakest precondition of a state set X with respect to the transition relation N. Some of the techniques for computing invariants are described briefly below.

Least Fixed Point of the Strongest Postcondition. The invariant computed here corresponds to the reachability state set. It is computed by starting with an initial symbolic representation of the initial state set given by the program. This set is successively enlarged by taking its image under the strongest postcondition operation until a fixed point is reached, i.e., no new elements are added to the set. We term this method LFP-SP. It yields a symbolic representation of the set of reachable states which is the strongest invariant. However, LFP-SP computation often does not terminate since the computation might not converge to a fixed point in a finite number of steps. Take, for example, a program that successively increments by one, a variable x that is initially zero. This program has a least fixed point, i.e., x is in the set of natural numbers, but the iterative computation does not converge.

For the *Twos* example, the LFP-SP computation does terminate with the desired invariant as seen in the calculation below.

$$\begin{aligned} Inv^0 &= (PC = inc \wedge B = 0 \wedge C = 0) \\ Inv^1 &= Inv^0 \vee (PC = dec \wedge B = 2 \wedge C = 0) \\ Inv^2 &= Inv^1 \vee (B = 0 \wedge C = 1) \\ Inv^3 &= (B = 0 \wedge C = 1) \vee (PC = dec \wedge B = 2 \wedge C = 0) \\ &= Inv^2 \end{aligned}$$

The resulting invariant easily implies the strengthened inductive invariant (4). The LFP-SP computation terminates precisely because the reachable state set is bounded. In more typical examples, approximation techniques based on *widening* will be needed to accelerate the convergence of the least fixed point computation.

Greatest Fixed Point of the Strongest Postcondition. The greatest fixed point iteration starts with the entire state space and strengthens it in each iteration

by excluding states that are definitely unreachable. This approach, which we call GFP-SP, yields a weaker invariant than the least fixed point computation. The GFP-SP computation also need not terminate. Even when it does terminate, the resulting invariant might not be strong enough. In the case of the program with single integer variable x that is initially zero and incremented by one in each transition, the GFP-SP computation returns the trivial invariant **true**. However the GFP-SP method has the advantage that it can be made to converge more easily than the LFP-SP method, and any intermediate step in the computation already yields a valid invariant.

The greatest fixed point invariant computation for *Twos* (ignoring the variable C) can be carried out as follows. Here $Inv^i(pc)$ represents the i iteration of the invariant for control state *pc*.

$$Inv^0(inc) = (B = 0 \vee B \geq -1) = (B \geq -1)$$
$$Inv^0(sub) = \mathbf{true}$$

$$Inv^1(inc) = (B \geq -1)$$
$$Inv^1(sub) = (B \geq 1 \vee B \geq -1) = (B \geq -1)$$

$$Inv^2(inc) = (B \geq -1)$$
$$Inv^2(sub) = (B \geq -1)$$

The invariant $B \geq -1$ is not all that useful since this information contributes nothing to the invariants that we wish to establish. Still, the GFP-SP method is not without value. It is especially useful for propagating known invariants. For example, if we start the iteration with invariant (1), then we can use the GFP-SP method to deduce that the strengthened invariant (4).

Greatest Fixed Point of the Weakest Precondition. Both LFP-SP and GFP-SP compute inductive invariants that are valid, whereas the method GFP-WP takes a putative invariant and strengthens it in order to make it inductive. The computation starts with a putative invariant S, and successively applies the weakest precondition operation wp(P)(S) to it. If this computation terminates, then either the resulting assertion is a strengthening of the original invariant that is also inductive, or the given invariant is shown to be invalid.

With the *Twos* example, the weakest precondition with respect to the putative invariant (1) yields the strengthened invariant (4).

2.5 Abstract Interpretation

Many of the invariant generation techniques are already examples of abstract interpretation which is a general framework for lifting program execution from the concrete domain of values to a more abstract domain of properties. Examples of abstract interpretation include sign analysis (positive, negative, or zero) of

variables, interval analysis (computing bounds on the range of values a variable can take), live variable analysis (the value of a variable at a control point might be used in the computation to follow), among many others.

We can apply an interval analysis to the *Twos* example. Initially, the interval for B is $[0,0]$ for $PC = inc$. This yields an interval of $[2,2]$ for B when $PC = dec$. In the next step, we have an approximation of $[0,2]$ for B when $PC = dec$, and $[0,0]$ when $PC = inc$. The next round, we get an approximation of $[-1,0]$ for the range of B when $PC = inc$, and $[0,2]$ for the range of B when $PC = dec$. At this point the computation converges, but the results of the analysis are still too approximate and do not discharge the invariant (1).

2.6 Property Preserving Abstractions

Since model checking is unable to cope with systems with infinite or large state spaces, abstraction has been studied as a technique for reducing the state space [12,29,35]. In data abstraction, a variable over an infinite or large type is reduced to one over a smaller type. The smaller type is essentially a quotient with respect to some equivalence relation of the larger type. For example, a variable ranging over the integers can be reduced to boolean form by considering only the parity (odd or even) of the numbers. *Predicate abstraction* is an extension of data abstraction that introduces boolean variables for predicates over a set of variables. For example, if x and y are two integer variables in a program, it is possible to abstract the program with respect to the predicates such as $x < y$, $x = y$. These variables are then replaced by boolean variables p and q such that p corresponds to the $x < y$ and q corresponds to $x = y$. Even though predicate abstraction introduces only boolean variables, it is possible to simulate a data abstraction of a variable to one of finite type by using a binary encoding of the finite type.

In general, an abstraction is given by means of a concretization map γ such that $\gamma(a)$ for an abstract variable a returns its concrete counterpart. In the case of the abstraction where $x < y$ is replaced by p and $x = y$ by q, $\gamma(a) = (x < y)$ and $\gamma(b) = (x = y)$. The more difficult direction is computing an abstraction $\alpha(C)$ given a concrete predicate C. The construction of α requires the use of theorem proving as described below.

There are also two ways of using abstractions in symbolic analysis. In one approach, the abstract reachability set [35,17] is constructed by the following iteration

$$ARG(P)(s) = lfp(\alpha(I_P) \vee \alpha \circ sp(P) \circ \gamma).$$

We can then check if p is an invariant of P by verifying $\gamma(ARG(P)) \supset p$.

A second way of using abstraction is by actually constructing the abstracted version of the program and the property of interest [8,16,37]. This can be more efficient since the program and property are usually smaller than the abstract reachability graph.

In the *Twos* example, the predicate abstraction is suggested by the predicates $B = 0$ and $B = 2$ in the putative invariant. The abstract transition system by replacing the predicate $B = 0$ by c and $B = 2$ by d is shown in Figure 2.

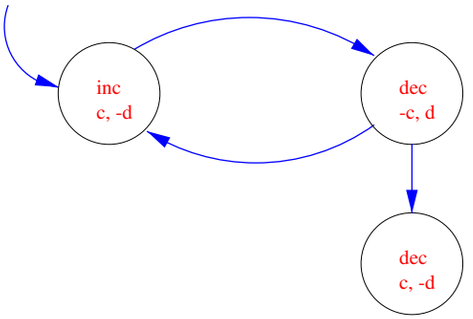


Fig. 2. Abstract *Twos*

The abstract transition system computed using predicate abstraction can easily be model checked to confirm that invariant (1) holds. The stronger invariant (4) can also be extracted from the reachable state space of the abstract transition system.

Predicate abstraction affords an effective integration of theorem proving and model checking where the former is used to construct a finite-state property-preserving abstraction that can be analyzed using the latter. The abstraction loses information so that a property can fail to hold in the abstract system even when its concrete counterpart is valid for the concrete system. In this case, the abstraction has to be refined by introducing further predicates for abstraction [8, 11].

3 SAL: A Symbolic Analysis Laboratory

We have already seen a catalog of symbolic analysis techniques. The idea of a symbolic analysis laboratory is to allow these techniques to coexist so that the analysis of a transition system can be carried out by successive applications of a combination of these techniques [6]. With such a combination of analysis techniques, one could envisage a verification methodology where

1. A cone-of-influence reduction is used to discard irrelevant variables.
2. Invariant generation is used to obtain small but useful invariants.
3. These invariants are used to obtain a reasonably accurate abstraction to a finite-state transition system.
4. Model checking is used to compute useful invariants of the finite-state abstraction.
5. The invariants computed by model checking over the abstraction are used propagated using invariant generation techniques.
6. This cycle can be repeated until no further useful information is forthcoming.

SAL provides a blackboard architecture for symbolic analysis where a collection of tools interact through a common intermediate language for transition

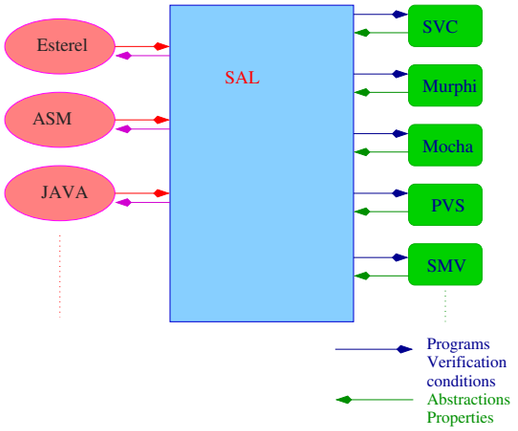


Fig. 3. The Architecture of SAL

systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate language and the analysis results are fed back to this intermediate level. In order to analyze systems that are written in a conventional source language, the transition system model of the source program has to be extracted and cast in the SAL intermediate language.[1] The model extracted in the SAL intermediate language essentially captures the transition system semantics of the original source program.

The SAL architecture is shown in Figure 3 The SAL architecture is constrained so that the different analysis tools do not communicate directly with each other, but do so through the SAL intermediate language. The interaction between the tools must therefore be at a coarse level of granularity, namely in terms of transition systems, their properties, and property-preserving transformations between transition systems. Allowing the tools to communicate directly to each other would require a quadratic number of different maps (for a given number of tools) between these analysis tools.

3.1 The SAL Intermediate Language

The intermediate language for SAL[2] serves as

1. The target of translations from source languages.
2. The source for translations to the input formats of different analysis tools.
3. A medium for communication between different analysis tools.

[1] We are currently working on a translator from a subset of Verilog to SAL, and another from a subset of Java to SAL.

[2] The SAL intermediate language was designed in collaboration with Prof. David Dill of Stanford, Prof. Tom Henzinger at UC Berkeley, and several colleagues at SRI, Stanford, and UC Berkeley.

The SAL intermediate language is based on languages and models such as SMV [30], Murphi [18], Reactive Modules [2], ASM [20], UNITY [15], and TLA [28], among others. The unit of specification in SAL is a context which contains declarations of types, constants, transition system modules, and assertions. A SAL module is a transition system unit. A basic SAL module is a state transition system where the state consists of *input*, *output*, *local*, and *global* variables, where

- An input variable to a module can be read but not written by the module.
- An output variable to a module can be read and written by the module, and only read by an external module.
- A local variable to a module can be read and written by the module, but is not read or written by the module.
- A global variable to a module can be read and written by the module as well as by an external module

A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition is of the form $x = \mathit{expression}$ or $x' = \mathit{expression}$, where x' refers to the new value of variable x in a transition. A definition can also be given as a selection of the form $x' \in \mathit{set}$ which means that the new value of x is nondeterministically selected from the value of of *set*. A guarded command is of the form $g \longrightarrow S$, where g is a boolean guard and S is a list of definitions of the form $x' = \mathit{expression}$ or $x' \in \mathit{set}$.

As in synchronous language such as Esterel [5] and Lustre [22], SAL allows *synchronous*, i.e., Mealy machine, interaction so that the new value of a local or output variable can be determined by the new value of a variable. Such interaction introduces the possibility of a causal cycle where each variable is defined to react synchronously to the other. Such causal cycles are ruled out by using static analysis to generate proof obligations demonstrating that such cycles are not reachable. The UNITY and ASM models do not admit such synchronous interaction since the new values of a variable in a transition are completely determined by the old values of the variables. SMV allows such interaction but the semantics is not clearly specified, particularly when causal cycles are possible. The Reactive Modules [2] language uses a static partial ordering on the variables that breaks causal loops by allowing synchronous interaction in one direction of the ordering but not the other. In TLA [28], two modules are composed by conjoining their transition relations. TLA allows synchronous interaction where causal loops can be resolved in any manner that is compatible with the conjunction of the transition relations is satisfied.

SAL modules can be composed

- *Synchronously*, so that $M_1 \| M_2$ is a module that takes M_1 and M_2 transitions in lockstep, or
- Asynchronously, so that $M_1 \,[]\, M_2$ is a module that takes an interleaving of M_1 and M_2 transitions.

There are rules that govern the usage of variables within a composition. Two modules engaged in a composition must not share output variables and nor should the output variables of one module overlap with the global variables of another. The modules can can share input and global variables, and the input variables of one module can be the output or global variables of the other. Two modules that share a global variable cannot be composed *synchronously*, since this might create a conflict when both modules attempt to write the variable synchronously. The rules governing composition allow systems to be analyzed modularly so that system properties can be composed from module properties [2].

The N-fold synchronous and asynchronous compositions of modules are also expressible in SAL. Module operations include those for hiding and renaming of variables. Any module defined by means of composition and other module operations can always be written as a single basic module, but with a significant loss of succinctness.

SAL does not contain features other than the rudimentary ones described above. There are no constructs for synchronization, synchronous message passing, or dynamic process creation. These have to explicitly implemented by means of the transition system mechanisms available in SAL. While these features are useful, their introduction into the language would place a greater burden on the analysis tools.

The SAL language is thus similar in spirit to Abstract State Machines [20] in that both serve as basic conceptual models for transition systems. However, machines described in SAL are not abstract compared with those in ASM notation since SAL is intended as a front-end to various popular model checking and program analysis tools.

4 Conclusions

Powerful automated verification technologies have become available in the form of model checkers for finite, timed, and hybrid systems, decision procedures, theorem provers, and static analyzers. Individually, these technologies are quite limited in the range of systems or properties they can handle with a high degree of automation. These technologies are complementary in the sense that one is powerful where the other is weak. Static analysis can derive properties by means of a syntactic analysis. Model checking is best suited for control-intensive systems. Theorem proving is most appropriate for verifying mathematical properties of the data domain. Symbolic analysis is aimed at achieving a synergistic integration of these analysis techniques. The unifying ideas are

1. The use of transition systems as a unifying model, and
2. Fixed point computations over symbolic representations as the unifying analysis scheme.
3. Abstraction as the key technique for reducing infinite-state systems to finite-state form.

Implementation work on the SAL framework is currently ongoing. The preliminary version of SAL consists of a parser, typechecker, causality checker, an invariant generator, translators from SAL to SMV and PVS, and some other tools. SAL is intended as an experimental framework for studying the ways in which different symbolic analysis techniques can be combined to achieve greater automation in the verification of transition systems.

Acknowledgments. Many collaborators and colleagues have contributed ideas and code to the SAL language and framework, including Saddek Bensalem, David Dill, Tom Henzinger, Luca de Alfaro, Vijay Ganesh, Yassine Lakhnech, Cesar Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, Eli Singerman, Mandayam Srivas, Jens Skakkebæk, and Ashish Tiwari.

References

1. Rajeev Alur and Thomas A. Henzinger, editors. *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, NJ, July/August 1996. Springer-Verlag.
2. R. Alur and T.A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
3. Nikolaj Bjørner, I. Anca Browne, and Zohar Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1):49–87, 1997.
4. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.
5. G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, and implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
6. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Muñoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, Hampton, VA, June 2000. NASA Langley Research Center. To appear.
7. Saddek Bensalem and Yassine Lakhnech. Automatic generation of invariants. *Formal Methods in Systems Design*, 15(1):75–92, July 1999.
8. Saddek Bensalem, Yassine Lakhnech, and Sam Owre. Computing abstractions of infinite state systems compositionally and automatically. In Hu and Vardi [26], pages 319–331.
9. Saddek Bensalem, Yassine Lakhnech, and Hassen Saïdi. Powerful techniques for the automatic generation of invariants. In Alur and Henzinger [1], pages 323–335.
10. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.
11. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Computer-Aided Verification*, Lecture Notes in Computer Science. Springer-Verlag, 2000. To appear.

12. Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
13. E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.
14. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables. In *5th ACM Symposium on Principles of Programming Languages*. Association for Computing Machinery, January 1978.
15. K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, MA, 1988.
16. M. A. Col on and T. E. Uribe. Generating finite-state abstractions of reactive systems using decidion procedures. In Hu and Vardi [26], pages 293–304.
17. Satyaki Das, David L. Dill, and Seungjoon Park. Experience with predicate abstraction. In Halbwachs and Peled [25], pages 160–171.
18. David L. Dill. The Murϕ verification system. In Alur and Henzinger [1], pages 390–393.
19. R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Proc. 15th Work. Protocol Specification, Testing, and Verification*, Warsaw, June 1995. North-Holland.
20. Yuri Gurevich. Evolving algebras 1993: Lipari guide. In Egon Börger, editor, *Specification and Validation Methods*, International Schools for Computer Scientists, pages 9–36. Oxford University Press, Oxford, UK, 1995.
21. S. M. German and B. Wegbreit. A synthesizer for inductive assertions. *IEEE Transactions on Software Engineering*, 1(1):68–75, March 1975.
22. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
23. C. A. R. Hoare. An axiomatic basis of computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
24. G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.
25. Nicolas Halbwachs and Doron Peled, editors. *Computer-Aided Verification, CAV '99*, volume 1633 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer-Verlag.
26. Alan J. Hu and Moshe Y. Vardi, editors. *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, Vancouver, Canada, June 1998. Springer-Verlag.
27. S. Katz and Z. Manna. Logical analysis of programs. *Communications of the ACM*, 19(4):188–206, April 1976.
28. Leslie Lamport. The temporal logic of actions. *ACM TOPLAS*, 16(3):872–923, May 1994.
29. 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*, 6:11–44, 1995.
30. Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1993.
31. Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Volume 1: Specification*. Springer-Verlag, New York, NY, 1992.

32. Zohar Manna and The STeP Group. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [1], pages 415–418.
33. A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science*, pages 46–57, Providence, RI, November 1977. ACM.
34. Hassen Saïdi. A tool for proving invariance properties of concurrent systems automatically. In *Tools and Algorithms for the Construction and Analysis of Systems TACAS '96*, volume 1055 of *Lecture Notes in Computer Science*, pages 412–416, Passau, Germany, March 1996. Springer-Verlag.
35. Hassen Saïdi and Susanne Graf. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer-Aided Verification, CAV '97*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83, Haifa, Israel, June 1997. Springer-Verlag.
36. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *4th ACM Symposium on Principles of Programming Languages*, pages 132–143, January 1977.
37. Hassen Saïdi and N. Shankar. Abstract and model check while you prove. In Halbwachs and Peled [25], pages 443–454.
38. Moshe Y. Vardi and Pierre Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1986.

Encoding Abstract State Machines in PVS

Angelo Gargantini[1] and Elvinia Riccobene[2]

[1] Dipartimento di Elettronica e Informazione - Politecnico di Milano - gargantini@elet.polimi.it
[2] Dipartimento di Matematica e Informatica - Università di Catania - riccobene@dmi.unict.it

Abstract. In this paper we show how the specification and verification system PVS (*Prototype Verification System*) can provide tool support for *Abstract State Machines* (ASMs), especially oriented towards automatic proof checking and mechanized proving of properties. Useful templates are presented which allow encoding of ASM models into PVS without any extra user's skill. We prove the transformation preserves the ASM semantics and provide a framework for an automatic tool, prototypically implemented, which translates ASM specifications in PVS. The ASM specification of the Production Cell given in [4] is taken as case study to show how to formalize multi-agent ASMs in PVS and prove properties.

1 Introduction

The *Gurevich's Abstract State Machines* (ASMs) [9] have been successfully used for design and analysis of complex hardware/software systems [3,2]. Through real case studies, ASMs have shown to be a practical method for rigorous system development and to meet the requirements, addressed by Heitmeyer in [10], that formal methods need to have "to be useful to practitioners": a user-friendly notation, useful, easy to understand feedback, integration into standard development process. ASMs use only the standard language and standard methods of programming and mathematics. They are models easy to read, understand and inspectable by the customer, but they are flexible to change, and precise and complete to match the designer's requirements. However, we believe that the success of a formal method in industrial field also depends on the availability of related automated tools helping the user during the process of specification and subsequent verification and validation. Therefore, machine support for the use of ASMs is extremely useful.

During the last years various investigations have been started in order to verify standard mathematical reasoning about ASMs by interactive or fully automated proof tools. Some encouraging results are already reported in literature (see discussion in [3]). Among theorem provers, PVS (*Prototype Verification System*) has been used in [7] to show the correctness of bottom-up rewriting specification for back-end compilers from an intermediate language into binary RISC processor code. Dold et al. state that "erroneous rules have been found using PVS and, through failed proof attempts, errors were corrected by inspection of the proof state". Mechanical checking of formal specifications and

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 303–322, 2000.

properties is useful both as a final confirmation of their correctness and for simplifying the process of discovering inconsistencies and proving properties. In fact, experience shows that even the most carefully crafted formal specifications and proofs, when done by hand, can still contain inconsistencies and other errors, and that such errors are discovered only using tools capable to perform some sort of analysis (also simply name checking, type consistency checking or other similar simple analysis). Proofs can also be so long and tedious that machine support can reduce the human effort.

In this paper, we show how PVS can provide tool support for ASMs, especially oriented towards automatic proof checking and mechanized proving of properties. PVS [13], developed by SRI, is a higher order logic specification and verification environment with a Gentzen-like deduction system. PVS provides *both* a highly expressive specification language and automation of proof steps.

Starting from the work of Dold et al. in [8,7] we develop suitable PVS theories to encode ASM models in PVS and we prove that our transformation preserves the semantics of ASMs. Our approach differs from that of Dold et al. in the way of encoding the ASM transition system. Our goal is to find a transformation from ASMs to PVS that preserves the layout of the original rule transition system. To achieve this goal we do not take any strong assumption as "only one rule enabled at a time", indeed we allow more rules to be simultaneously executed. Therefore, we do not demand that the user should transform the whole system of rules in one meta rule, transformation which can require skill and ingenuity, especially in case of multi-agent models. Instead, we propose an algorithmic approach of transformation which keeps the set of transition rules as a set of different rules.

We provide useful templates for helping the user during the encoding phase, and useful proof schemes for requirements verification. The suggested encoding scheme is mechanizable and we provide a framework for a tool, that we prototypically implemented, supporting automatic transformation in PVS of ASM models given in the ASM-SL language [5].

Our PVS encoding of ASMs has been tested for multi agent models. We report here the PVS specification of the Production Cell given in [4], and we discuss the results of the mechanized proofs of safety and liveness properties. Several proofs are obtained with assumptions, usually regarding the physical environment behavior, that are implicit in the formal description but need to be added in the machine supported approach. Thanks to the PVS features, we formulate such assumptions in a declarative way instead of using the operational way required by the model checking approach used by Winter in [14] to show the correctness of the Production Cell specification.

The article is organized as follows. The problem of encoding ASMs in PVS is discussed in Sections 2.1-2.3. Section 2.4 shows how to validate specifications using PVS. Sections 2.5 and 2.6 present proof schemes to support verification of invariant properties and trace properties. Section 3 contains the PVS specification of the Production Cell case study and the results of the verification task. In Section 4 we discuss our work in comparison with related results, and we conclude outlining future research directions.

2 An Encoding of ASMs in PVS

In this section we present our encoding of ASMs in PVS. It consists of a set of PVS theories, hierarchically organized, which define types and functions modeling ASM universes and rules, and of a set of PVS strategies defined in order to simplify the proof conduction. Universe and function representation is based on the work of Dold et al. [8,7]. PVS encoding of rules and dynamics of ASMs is significantly changed.

2.1 Abstract State

An ASM *state* models a(n abstract) machine state, i.e. the collection of elements the machine "knows", and the functions and predicates it uses to manipulate such elements. Mathematically a *state* is defined as a structure which consists of a collections of domains (sets, also called *universes*, each of which stands for a particular type of elements) with arbitrary functions and relations defined on them. **Universes representation.** Universes have different encoding depending on their being static or dynamic:

- A *static universe* U, i.e. a universe which does not change during computation, is encoded as uninterpreted type `U:TYPE`.
- A *dynamic universe* U, i.e. a universe which may grow during computation, is encoded as a PVS set `U:setof[T]` of elements of type T.
  As elements can not be added to PVS types, only such encoding allows to expand universes. To this purpose, the functions *add* and *union*, (pre-)defined in the PVS theory `sets`, can be used.

Remark 1. The concept of type in PVS is very different from the usual meaning of type in programming languages. PVS types abstractly represent entities with common operations. There are only a few built-in PVS types such as numbers, integers and boolean. Other types may be defined either as uninterpreted types, or through the construct `DATATYPE`, or in terms of already defined types by usual type constructors (as lists, records, etc.).

It is possible to encode the superuniverse itself as an uninterpreted type `S` and define every universe `U` as a subset of `S` – as already noted by Dold in [8]–, or as uninterpreted subtype (with the same meaning as subset). However, in the last case it is necessary to introduce axioms to guarantee properties on sets, as, for example, disjointness between universes, so not exploiting the strong and powerful type system of PVS. For this reason we suggest to encode universes as uninterpreted types whenever possible, and use the construct `setof` only for dynamic universes.

Functions and states. ASM *basic* functions are classified in *static* functions which remain constant, and *dynamic* functions which may change interpretation during computation. Dynamic functions are furthermore classified in *controlled*, *monitored* and *shared*. Controlled functions occur inside function updates and

can change by application of transition rules. Monitored functions can change only due to the environment. Shared functions can change by application of a transition rule but also by some other agents; they formalize the interactions in case of multi-agent computations and combined updates of locations. Functions defined in terms of the basic ones are called *derived*.

We encode functions almost as proposed in [8].

- *Static functions* are encoded as usual PVS functions.
- *Monitored functions* are modeled as PVS functions adding the type `ENV` to the arguments of their domain. `ENV` is an uninterpreted type which represents the external environment, that is everything outside the system's control. A boolean dynamic monitored variable is therefore encoded as a function from `ENV` to `bool`, whereas a generic monitored function $f : S \rightarrow T$ is translated into the function `f:[ENV,S->T]`.
  Our way of interpreting `ENV` is different from that proposed in [8] where `ENV` is the record of all monitored functions. Our view of `ENV` has the advantage of allowing a designer to easily add and remove monitored functions.
- *Controlled functions* are regarded as fields of a record `CTRLSTATE` which represents the controlled part of an ASM state. Therefore, if $f : A \rightarrow B$ and $g : Int \rightarrow Bool$ are controlled functions, the controlled part of the state is defined as `CTRLSTATE: TYPE [# f:[A->B], g:[int->bool] #]`.
- *Shared function* are defined as controlled functions. In a multi-agent ASM, the controlled part of a "global" state is viewed as the collection of all controlled agent's states. Therefore, shared functions, as controlled by at least one agent, are defined as components of the `CTRLSTATE`.
- *Derived function* are encoded as functions defined in terms of their function components in a straightforward way from their ASM specification.

An ASM *state* is compound of its controlled part and the environment. It is defined as the record `STATE: TYPE [# env: ENV, ctrl: CTRLSTATE #]`.

2.2 ASM Rules

ASM *(transition) rules* model the actions performed by the machine to manipulate elements of its domains and which will result in a new state. A *transition rule* R_i has the general form:

$$R_i : \textbf{if } cond_i \textbf{ then } update_i$$

$cond_i$, also called the *guard* of the rule, is a statement expressing the condition under which R_i must be applied; $update_i$ consists of finitely many *functions updates*:

$$f(t_1, \ldots, t_n) := t$$

which are executed simultaneously. f is an arbitrary n-ary function and $t_1, ..., t_n$ is the tuple of arguments at which the value of the function is set to t.

How to serialize the parallel rule application. In this section we tackle the problem of finding an algorithm to serialize firing of ASM rules which guarantees the semantics of their parallel application as stated in [3]:

> An ASM M is a finite set of rules [...]. Applying one step of M to a state $\mathcal{A}$ produces as next state another algebra $\mathcal{A}'$, of the same signature, obtained as follows.
> First evaluate in $\mathcal{A}$, using the standard interpretation of classical logic, all the guards of all the rules of M. Then compute in $\mathcal{A}$, for each of the rules of M whose guard evaluates to true, all the arguments and all the values appearing in the updates of this rule. Finally replace, simultaneously for each rule and for all the locations in question, the previous $\mathcal{A}$ function value by the newly computed value.

Let $\{R_1, ..., R_n\}$ be a set of rules. The naive approach of computing the next state $\mathcal{A}'$ of the current state $\mathcal{A}$ by applying rules R_i, $i = 1 \dots n$, sequentially,[1] does not guarantee the ASM semantics for two reasons: (a) guards $cond_i$ of R_i are not evaluated in the same state $\mathcal{A}$ forall i; (b) terms occurring in the left and right sides of function updates of R_i are not all evaluated in $\mathcal{A}$. The following example may help to convince that this approach is wrong.

Example. Let $\mathcal{A} = (x \to 0, y \to 1, z \to 2)$ be the current state consisting of three integer variables *x, y, z*, and $\{R_1, R_2, R_3\}$ be the transition system with rules defined as follows:

$$R_1 : \textbf{if } x = 0 \textbf{ then } y := 5 \qquad R_2 : \textbf{if } y = 5 \textbf{ then } x := 2 \qquad R_3 : z := x + y$$

By sequential rules application, it results:

$$\mathcal{A} : \begin{pmatrix} x \to 0 \\ y \to 1 \\ z \to 2 \end{pmatrix} \overset{R_1}{\to} \begin{pmatrix} x \to 0 \\ y \to \boxed{5} \\ z \to 2 \end{pmatrix} \overset{R_2}{\to} \begin{pmatrix} x \to \boxed{2} \\ y \to 5 \\ z \to 2 \end{pmatrix} \overset{R_3}{\to} \begin{pmatrix} x \to 2 \\ y \to 5 \\ z \to \boxed{7} \end{pmatrix} : \mathcal{A}'$$

whereas by correct computation would be $\mathcal{A}' = (x \to 0, y \to 5, z \to 1)$.
An **algorithm** which overcomes the errors of the naive attempt is given below.

Let $\{R_1, ..., R_n\}$ be a set of transition rules, R_i of the form **if** $cond_i$ **then** $update_i$ with $update_i$ a sequence of function updates $f(t_1, t_2, \dots, t_n) := t$, and $\mathcal{A}$ be the current state.
The algorithm can be stated as follows:

1. For all rules R_1 ,..., R_n, mark R_i if enabled at the state $\mathcal{A}$.
2. For all i such that marked R_i, evaluate in $\mathcal{A}$ all terms $t_1, t_2, \dots, t_n, t$ occurring in function updates $f(t_1, t_2, \dots, t_n) := t$ of $update_i$.
3. For all i such that marked R_i, sequentially perform the function updates in $update_i$ using the values computed at step 2.

The proposed algorithm initially evaluates all the guards $cond_i$ in $\mathcal{A}$ and only afterwards, sequentially, performs all the function updates. Applying the algorithm to the above example we get the following correct transformation:

[1] I.e., first evaluate the guard of R_1 and, if it is true, perform the updates defined in R_1 to obtain $\mathcal{A}_1$; then apply R_2 to $\mathcal{A}_1$ and obtain $\mathcal{A}_2$; and so on till computing the final state $\mathcal{A}_n$ to be taken as $\mathcal{A}'$.

$$\mathcal{A}: \begin{pmatrix} x \to 0 \\ y \to 1 \\ z \to 2 \end{pmatrix} \overset{R_1}{\to} \begin{pmatrix} x \to 0 \\ y \to \boxed{5} \\ z \to 2 \end{pmatrix} \overset{R_2}{\to} \begin{pmatrix} x \to 0 \\ y \to 5 \\ z \to 2 \end{pmatrix} \overset{R_3}{\to} \begin{pmatrix} x \to 0 \\ y \to 5 \\ z \to \boxed{1} \end{pmatrix} : \mathcal{A}'$$

Rule R_2 does not affect the state $\mathcal{A}$ because not enabled, and R_3 correctly computes the new value of z using the values of x and y in $\mathcal{A}$.

Inconsistent updates. If a location is updated to different values by simultaneously firing rules, then we speak of "inconsistent updates". Although the location value should remain unchanged due to the non-execution of inconsistent updates [9], according to our algorithm, the location would take the last value assigned to it. For this reason, at the moment, we are able to consider only consistent ASM models. It would be possible to discover inconsistent updates through simulation of suitable test cases. However, simulation can never guarantee absence of inconsistency. Applying a formal verification technique similar to that used for proving invariant properties in Section 2.4, we might discover inconsistent updates. Checking inconsistency of ASM specifications is a problem still under investigation. Note that the problem of firing inconsistent updates is not taken in consideration in [8,7].

How to deal with extend. The proposed algorithm does not give the expected result when dealing with combination of rules which simultaneously add elements to a same set by the construct **extend**. This problem arose trying to encode in PVS the ASM model of crypto-protocol given in [1]. In this model, different agents can send at the same time messages into the TRAFFIC (a set of messages) representing the network channel common to all agents.

To be convinced why **extend** must be dealt with particular attention, consider the following example. Let $\mathcal{A}$ be a state where a set Δ has value $\emptyset$. The following rules are enabled in $\mathcal{A}$:

$$R_1\text{: } \textbf{if } \Delta = \emptyset \textbf{ then extend } \Delta \textbf{ with } a$$
$$R_2\text{: } \textbf{if } \Delta = \emptyset \textbf{ then extend } \Delta \textbf{ with } b$$

Encoding "**extend** Δ **with** u" as a function update of the form $\Delta := \Delta \cup \{u\}$, by our algorithm, we get the following transformation:

$$\mathcal{A} : (\Delta \to \{\}) \overset{R_1}{\to} (\Delta \to \{a\}) \overset{R_2}{\to} (\Delta \to \{b\}) : \mathcal{A}'$$

However, as both rules are applicable, according to the correct semantics of **extend**, we expect the result $\Delta = \{a, b\}$ in the final state $\mathcal{A}'$.

The algorithm needs to be adjusted to deal with extend. A correct solution should (a) evaluate expressions in the current state; (b) fire updates not involving **extend** against the current state; (c) if a rule R_i contains an update **extend** U **with** u, add u to the set U interpreted not in the current state, but in the state obtained after firing all rules preceding R_i (otherwise we could loose updates of U by other possible **extend**s of previous rules).

The **revisited version of the algorithm** follows:

1. For all rules R_1 ,..., R_n, mark R_i if enabled at the state $\mathcal{A}$.
2. For all i such that marked R_i, evaluate in $\mathcal{A}$ all terms occurring in function updates of $update_i$ with exception of those representing universes as arguments of **extend**.
3. Let, unless renaming, $\{R_1, ..., R_m\}$ be the set of all marked rules. Assume $s_0 = \mathcal{A}$. Sequentially, compute s_i from s_{i-1}, $i = 1 \ldots m$, performing function updates of $update_i$ of rule R_i in s_{i-1}: use values computed at step 2 for all terms except those representing universes within **extend** which must be evaluated in s_{i-1}. The final state s_m be $\mathcal{A}'$.

Remark 2: *Reserve* and multisets. According to the ASM semantics, elements added to extended universes are "new" and imported from *Reserve*. To guarantee that, Dold et al. introduce a predicate `new` to control an element does not already belong to the set to be extended, and a function `sort_update` which extends a set by new elements. However, PVS does not allow sets to contain different occurrences of the same element, and even Dold's formalization does not avoid such a problem. When an ASM specification requires a universe to contain multiple occurrences of a same element, we suggest to encode the universe as a multiset instead of a set. Keeping in mind that in PVS a set of elements of type `T` is a function from `T` to `bool`, as natural extension, a multiset can be encoded as a function from `T` to natural numbers: the value of the function represents the number of occurrences of its argument in the multiset. Functions and predicates on sets, like `add`, `member`, `emptyset`, etc., must be redefined. The PVS theory to model multisets follows:

```
multisets [T: TYPE]: THEORY
  BEGIN
   multiset: TYPE = [T-> nat]
   x, y: VAR T
   a: VAR multiset
% an element x is member of a only if the occurrences
% of x in a are greater than 0
   member(x, a): bool = a(x) > 0
   empty?(a): bool = (FORALL x: NOT member(x, a))
   emptyset: multiset = lambda x: 0
   nonempty?(a): bool = NOT empty?(a)
% the function ‘‘add’’ returns the same multiset modified
% increasing by one the number of occurrences of the added
% element
   add(x, a): (nonempty?) = lambda y:
             if x = y then a(y)+ 1 else a(y) endif
  END multisets
```

Remark 3. The rule encoding proposed in [8,7] is based on the (strong) assumption that rules "concern distinct (disjoint) cases". As a consequence, one

may infer that (a) at every step only one rule is enabled, and therefore (b) all rules "may be combined into a single state transition function, defined by a case construct or nested conditionals". Such a single transition function is called "one-step interpreter" by Dold et al. This assumption of disjoint cases is restrictive, since in many applications more then one rule might fire at a time. In case of possible simultaneous firing of many rules, a unique transformation function might still model the ASM dynamics as well, however we prefer keeping the set of transition rules as set of distinct rules, without forcing the user to define an equivalent transformation function from the current state to the next one. The advantages are two: first stylistic, because we preserve the ASMs structure, and second practice, because writing one equivalent global transformation function requires skill and ingenuity. Furthermore, due to the assumption that only one rule can be applied in one step, the one-step interpreter is absolutely not suitable in case of multi-agent ASMs.

Remark 4. Another possible approach for rule encoding would be to define the value of every variable and function (location) in the next step by means of axioms. For example, the update $f(x) := t$ would be translated into an axiom of the form `update : AXIOM f(next(s))(x) = t`, where `next(s)` stands for the next state. In this way it would not be necessary to introduce the state as a record and we could consider the state as simple uninterpreted type. This approach is similar to that taken by [14] to translate ASMs in SMV. However, this translation does not preserve the original form of the rules and problems rise when a rule consists of more than one function update, or when more rules update the same function (location). In both cases, a complex transformation is needed as described in detail in [14].

2.3 Implementing ASM Rules in PVS

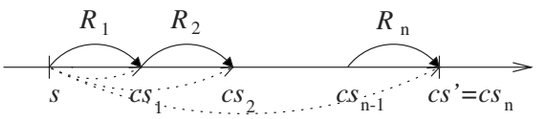
In this section we show how the proposed algorithm has been implemented in PVS. Keep in mind that in our encoding a *state* consists of a controlled part (`CTRLSTATE`) modifiable by the rules, and the environment (`ENV`).

Rule encoding. We consider a rule as a function `Ri(current,intCtrl)`. The first argument `current: STATE` represents the current state s (controlled part and environment), and it is used to evaluate guards (step 1 of the algorithm) and compute terms (step 2 of the algorithm). The second argument `intCtrl: CTRLSTATE` represents an intermediate controlled state which is the result of applying the previous $i-1$ rules, and is used to compute those updates extending sets (step 3 of the algorithm). `Ri(current,intCtrl)` yields a new controlled (intermediate) state.
In PVS a rule is defined as `Ri:[STATE,CTRLSTATE -> CTRLSTATE]`.

Computation of the next state by composition of rules. To compute the next state of a current one by application of a set of rules, we define the function `next` as explained below.

Let $R_1, R_2, \ldots, R_n$ be a set of rules, and s the current state (its controlled part be cs). The controlled part cs' of the next state s' is inductively defined as follows:

1. $cs_0 = cs$
2. for $i = 1, \ldots, n : cs_i = R_i(s, cs_{i-1})$
3. $cs' = cs_n$



This algorithm defines only how to compute the controlled part of s' since the rules do not change the monitored part (i.e. the environment).

The set of all ASM rules is encoded as a list `rules` of functions from state and controlled state to a controlled state:

```
rules: list[[STATE,CTRLSTATE->CTRLSTATE]]
```

The application of the list `rules` using as current state `s0` and as intermediate controlled state `cs_i`, is given by the following recursive definition[2]:

```
apply(s0,cs_i,rules) : recursive CTRLSTATE =
 if null?(rules)    then cs_i
 else apply(s0,car(rules)(s0,cs_i),cdr(rules))
endif measure length(rules)
```

The controlled part of the next state of `s` is defined applying the list `rules` of all ASM rules, and taking `s` as initial current state and its controlled part as initial intermediate controlled state:

```
nextCtrlState(s:STATE): CTRLSTATE = apply(s,s‘ctrl,rules)
```

The monitored part (the environment) of the next state of `s` is defined by the following function returning a random value for `ENV`:

```
nextEnv(s:STATE) : ENV
```

The next state of `s` is the composition of the two next (controlled and monitored) parts:

```
next(s:STATE): STATE =
(#env:= nextEnv(s), ctrl:= nextCtrlState(s) #)
```

[2] `car` and `cdr` are built-in PVS functions yielding head and tail of a list, respectively.

Templates for rules encoding. We report here the two templates for rules encoding. They distinguish between rules with and without **extend**ing updates. Let $R_i \equiv$ **if** $cond_i$ **then** $update_i$ be a transition rule.

1 If $update_i$ is a sequence of function updates of the form $f(t_1, \ldots, t_n) := t$, then R_i is translated in PVS as follows:

```
Ri(current, intCtrl) : CTRLSTATE =
 IF cond_i(current) THEN intCtrl
           WITH [f:=f(intCtrl) WITH [(t1,...,tn):=t]]
 ELSE intCtrl    ENDIF
```

If the guard `cond_i` is true in the `current` state then the updated intermediate controlled state `intCtrl` is returned; otherwise the state `intCtrl` is returned unchanged (all terms `ti` and `t` are computed in the `current` state).

2 If $update_i$ has the form **extend** *Delta* **with** *alpha*, then R_i is translated in PVS as follows:

```
Ri(current, intCtrl) : CTRLSTATE =
IF cond_i(current) THEN intCtrl
          WITH [Delta := add(alpha,Delta(intCtrl))]
ELSE intCtrl   ENDIF
```

If the guard `cond_i` is true, the element *alpha* is added to *Delta* evaluated in `intCtrl`; otherwise `intCtrl` is returned unchanged.

The following examples may help to better understand rules encoding.

Example 1. Let the ASM rules be

R_1 : **if** $x = 0$ **then** $y := 5$ $\quad R_2$: **if** $y = 5$ **then** $x := 2$ $\quad R_3 : z := x + y$

The controlled state is defined as a record of three variables:

```
CTRLSTATE: TYPE = [# x: int,  y: int,  z: int #]
```

The three rules are defined as follows:

```
R1(current, intCtrl) : CTRLSTATE =
 IF x(current) = 0    THEN intCtrl  WITH [y := 5]
 ELSE intCtrl  ENDIF
R2(current, intCtrl) : CTRLSTATE =
 IF y(current) = 5    THEN intCtrl  WITH [x := 2]
 ELSE intCtrl  ENDIF
R3(current, intCtrl) : CTRLSTATE =
 intCtrl  WITH [z := x(current) + y(current)]
```

Example 2 (with sets).

R_1: **if** $\Delta = \emptyset$ **then extend** Δ **with** a

R_2: **if** $\Delta = \emptyset$ **then extend** Δ **with** b

Assuming that elements of the set Δ belong to a certain type `elementType`, and a and b are two constants of that type:

```
elementType : TYPE
a,b : elementType
```

the controlled state is defined as a record containing only the set Δ:

```
CTRLSTATE: TYPE = [# Delta: SETOF[elementType]#]
```

The two rules become:

```
R1(current, intCtrl) : CTRLSTATE =
 IF empty?(Delta(current))  THEN intCtrl
           WITH [Delta := add(a,Delta(intCtrl))]
 ELSE intCtrl  ENDIF
R2(current, intCtrl) : CTRLSTATE =
 IF empty?(Delta(current))  THEN intCtrl
           WITH [Delta := add(b,Delta(intCtrl))]
 ELSE intCtrl  ENDIF
```

Non determinism. In ASM non-determinism can be expressed using the constructor **choose** which allows to fire a rule $R(x)$ choosing randomly an x satisfying given conditions:

choose x **in** U **s.t.** $g(x)$
$R(x)$

In PVS **choose** can be encoded by a monitored function from the environment to the subset of U consisting of elements satisfying the condition $g(x)$.

```
chooseX : [ENV-> {x:U|g(x)}]
```

The subcase **choose** x **in** U is encoded by `chooseX : [ENV->U]`. The non-determinism is captured by leaving undefined (without no specified mathematical low) the function `chooseX`. Therefore, for a given environment `e`, the value of `chooseX(e)` in `{x:U|g(x)}` is not determined.

2.4 Validating Specifications through Simple Proofs

After having specified system universes, functions and rules, the designer should check whether the specification is correct or not, i.e. if it meets users' needs and if it satisfies all the desired properties (requirements).

The first step in this direction is to check some possible behaviors of the system as specified and compare them with the desired behaviors. This approach is followed by ASM simulators (like ASM Workbench and ASM-Gofer). A similar approach might be followed in our encoding, probing the specification by means of "formal challenges". With this term we mean putative theorems, i.e. properties that should be true if the specification is correct. The designer should start from formally specifying and proving very simple statements and then gradually prove more complex properties. Only at the end he/she should try to prove the complete requirements.

In our Example 1 a very simple property is: "if in state s x is 0, y is 1 and z is 2, then in the next state x should be equal to 0, y to 5 and z to 1". It can be encoded as the following lemma:

```
prop1: lemma
s`ctrl= (#x:=0,y:=1,z:=2#) => next(s) = (#x:=0,y:=5,z:=1#)
```

We have proved the lemma `prop1` simply using the PVS decision procedures of rewriting and symbolic execution. More complex and complete properties might contain quantification on system quantities and, therefore, to be proven they might need the use of more advanced PVS strategies.

The main goal of these short proofs is to gain a deeper understanding of the system, to become confident in the correctness of the proposed formal description, and discover possible errors or faults as early as possible, since it is widely acknowledged that the cost of correcting specification errors is order of magnitudes higher in the later stages of the life cycle of the system (like during testing or even during its normal functioning). At the end the designer should be able to formally state and prove the actual requirements. The verification of system requirements is the subject of the following sections.

2.5 Using Induction to Prove Invariants

In mathematics induction is widely used to prove properties that hold for every natural number. In formal models that describe system behavior as a sequence of states (as the ASM approach does), the same scheme can be used to prove system *invariants*, i.e. properties holding in every state. In this case induction is based on the following theorem:

Theorem 1. *Let S_0 be the set of all initial states and $P(s)$ a property of the state s. If*

(i.) $P(s_0)$ holds $\forall s_0 \in S_0$;

(ii.) $P(s) \rightarrow P(s')$, $\forall s, s'$ states such that $s' = next(s)$

then P is an "invariant".

In our encoding we have defined and proved theorem 1 as follows:

```
induction: THEOREM
       (forall(s:STATE): P(s)=> P(next(s)) and
       (forall(s:(init)): P(s)) implies INV(P)
```

where `(init)` denotes the set of initial states and `P` the property to prove as invariant (`INV(P)` means that `P` is true in the initial state and in every reachable state).

This theorem, along with an "ad hoc" strategy that we have defined in the file `pvs-strategies`, provides an induction scheme that can be used to prove invariants in ASM models.

2.6 Trace Properties

A *trace* is an (infinite) sequence $s_0, s_1, \ldots, s_n, \ldots$ of states – any state s_i should be considered as compound of the controlled part and the environment – satisfying the property that: (a) s_0 is a valid initial state, and (b) for every pair of subsequent states s_i, s_{i+1} it holds that s_{i+1} is the next state of s_i.

In our encoding a trace is formalized as a sequence of states satisfying the property of being a *trace*:

```
trace: TYPE = {x : sequence[STATE] | member(first(x),init)
                  and forall n: nth(x,n+1) = next(nth(x,n))}
```

Trace properties are properties on traces, i.e. properties which are expressed and proved in terms of traces. The two most common types of trace properties are:

- properties holding in every state of every trace (*always*), or
- properties holding at least in one state in every trace (*eventually*).

We can express in PVS that a property *stateProp* holds in every state of the trace t by the following predicate:

```
always(t,stateProp): bool = FORALL n: stateProp(nth(t,n))
```

We have proved the equivalence between this approach based on traces and that based on invariants by proving the following lemma

Lemma 1. *"stateProp" is an invariant iff it always holds in every trace of the system.*

In PVS:
```
equivalence: LEMMA
       INV(stateProp) <=> forall t: always(t,stateProp)
```

We express in PVS that a property *reachProp* holds in a state of the trace t by the following predicate:

```
eventually(t,reachProp): bool = EXISTS n: reachProp(nth(t,n))
```

The *always* property is normally used to express "safety" requirements ("nothing bad will never occur"): properties that must be true in every state. The *eventually* property normally expresses "liveness" properties ("something good will eventually happen") modeling requirements that must be eventually true.

3 A Case Study: Production Cell

In this section we explain this novel use of PVS as tool support for ASMs using as case study the Production Cell model given in [4]. The main purpose of this section is not to show that the Production Cell specification of Börger and Mearelli satisfies safety and liveness properties. That has been already proved in [14] by means of the model checker SMV. We simply like to show, through a concrete example, how to apply our method of encoding ASM specifications in PVS, and how to mechanize proofs.

3.1 A Brief Introduction of the Production Cell Case Study

The production cell control problem was posed in [11] as case study derived from "an actual industrial installation in a metal-processing plant in Karlsruhe" to obtain a "realistic, comparative survey" for testing "the usefulness of formal

methods for critical software systems and to prove their applicability to real-world examples" [11]. Börger and Mearelli propose a solution of the production cell control problem in [4], and show how to integrate the use of ASMs into a complete software development life cycle.

> *... the production cell is composed of two conveyor belts, a positioning table, a two-armed robot, a press, and a traveling crane. Metal plates inserted in the cell via the feed belt are moved to the press. There, they are forged and then brought out of the cell via the other belt and the crane. [11]*

The system is specified "as a distributed ASM with six modules, one for the *agents*" – the Feed Belt, the Robot, the Press, the Deposit Belt, the Traveling Crane, the Elevating Rotary Table – "composing the production cell, and working together concurrently where each of the component ASMs follows its own clock. Each of the agents represents a sequential process which can execute its rules as soon as they become enabled. The sequential control of each agent is formalized using a function currPhase: Agent → Phase which yields at each moment the *current phase of the agent*"[4]. We refer the reader to [4] for further details.

3.2 The PVS Specification

For convenience, we report below the signature and the module of the Ground-CELL Program for the Feed Belt (FB).

Monitored function: PieceInFeedBeltLightBarrier
Shared function: TableLoaded (between FB and the elavating rotary table ERT)
Derived functions:
TableInLoadPosition ≡ currPhase(ERT) = StoppedInLoadPosition
TableReadyForLoading ≡ TableInLoadPosition and not TableLoaded
Controlled functions: FeedBeltFree,
currPhase(FB) ∈ {NormalRun,Stopped,CriticalRun}.
Module:

```
FB _NORMAL.
    if currPhase = NormalRun and PieceInFeedBeltLightBarrier
    then  FeedBeltFree:= True
          if TableReadyForLoading then currPhase:= CriticalRun
                                  else currPhase:= Stopped
FB_STOPPED.
    if currPhase = Stopped and TableReadyForLoading
    then currPhase:= CriticalRun

FB_CRITICAL.
    if currPhase = CriticalRun and PieceInFeedBeltLightBarrier
    then  currPhase:= NormalRun
          TableLoaded:= True
```

Initialization: currPhase = NormalRun, FeedBeltFree = True,
PieceInFeedBeltLightBarrier = False

We now report the PVS encoding of the Feed Belt specification. To describe the Feed Belt we define a type containing all the possible values of the feed belt phase:

```
FBPhase : TYPE = {NormalRun,Stopped,CriticalRun}
```

The controlled function `FeedBeltPhase` is then included as component of the record `CTRLSTATE` which represents the controlled part (controlled and shared functions) of the (global) state:

```
CTRLSTATE : TYPE =
 [#   FeedBeltPhase : FBPhase,
      FeedBeltFree  : bool, % controlled by the FB
      TableLoaded   : bool, % controlled by the FB and ERT
      ...#]
```

The dots are replaced by the controlled part of the other five agents, that we skip for the sake of conciseness. Monitored variable is defined as function from the environment to its domain as:

```
PieceInFeedBeltLightBarrier : [ENV->bool]
```

Starting from definitions of monitored and controlled functions, derived functions are defined as:

```
TableInLoadPosition(s:CTRLSTATE) : bool =
     ERTPhase(s) = StoppedInLoadPosition
TableReadyForLoading(s:CTRLSTATE) : bool =
     TableInLoadPosition(s) and not TableLoaded(s)
```

The initial state is modeled by a predicate over the states (we report only the part concerning FB):

```
init(s:STATE) : bool =
FeedBeltPhase(s)= NormalRun and FeedBeltFree(s)
and not PieceInFeedBeltLightBarrier(s) ...
```

For the rules we report only the example of the FB_NORMAL rule:

```
FB_NORMAL(current,intCtrl): CTRLSTATE =
if FeedBeltPhase(current) = NormalRun and
   PieceInFeedBeltLightBarrier(current)
then intCtrl with
    [FeedBeltFree := true,
    FeedBeltPhase := if TableReadyForLoading(current)
         then CriticalRun
         else Stopped
         endif]
else intCtrl
endif
```

3.3 The Safety Properties

Using PVS we have proved all the safety properties (for the Feed Belt, the Robot, the Press, the Deposit Belt, and the Traveling Crane) of the Production Cell as given in [4]. For some of these properties, the encoding in PVS and the proof are straightforward. Others require some user effort and skill. In order to discuss the degree of interaction necessary for proving in PVS the requirements of the Production Cell case study, we present some selected examples of proved properties having different degree of complexity.

The **Feed Belt Safety Property:** *the feed belt does not put metal blanks on the table if the latter is already loaded or not stopped in loading position.* It has been quickly encoded considering that the feed belt puts metal blanks only when it is in the CriticalRun phase[3]:

```
FeedBeltSafety: theorem
FeedBeltPhase(s) = CriticalRun
  => ElevatingRotaryTablePhase(s) = StoppedInLoadPosition
     and not TableLoaded(s)
```

The proof of this property, reported below, is immediate (as also its hand proof in [4]):

```
("" (ASM-INDUCT)
      (("1" (COMPUTE-NEXT) (GRIND))
       ("2" (TYPEPRED "is!1") (GRIND))))
```

(ASM-INDUCT) applies the induction theorem presented in Section 2.5. By induction the proof is split in two parts: the induction step (proved by the branch "1") and the initial state (branch "2"). (COMPUTE-NEXT) is a strategy defined in our encoding and expands the definitions of the next state and the rules. (GRIND) is a PVS command that rewrites remaining definitions, splits the cases and applies the decision procedures of PVS. (TYPEPRED "is!1") recalls the type definition of the initial state "is!1". The last (GRIND) expands the definition of initial state and applies the PVS decision procedures.

This proof case shows that the user effort to write properties and obtain relative proofs might be very low. According to our experience, all simplest properties whose proof do not involve assumptions about the environment, require a minimal user interaction and their proofs can be performed using induction, case splitting, decision procedures, and rewriting rules. However, many other properties have to be proved taking in consideration the interaction of the system with the environment. In these cases, proving properties might require greater user effort and skill. As example, consider the first part of the **Press Safety Property 1:** *the press is not moved downward if it is in its bottom position.* Its PVS encoding is straightforward:

[3] In this lemma and in the following ones, `s` must be considered as universally quantified over the set of all reachable states from an initial state.

```
PressSafety1a: theorem
     PressBottomPosition(s) =>
             not PressMotorDown(Press(next(s)))
```

To prove this property we have to introduce an assumption about the monitored function `BottomPosition` asserting that *if the press is closed for forging, then it is not already in the bottom position*:

```
notBottom: axiom
PressPhase(s)= ClosedForForging => not BottomPosition(s)
```

This is an obvious implication considering how the system works, but we have to explicitly state that by means of an axiom. Upon introducing the `notBottom` axiom, the proof of the Press Safety Property 1 is obtained applying induction, expanding the definitions and applying the PVS decision procedures. For other properties we introduce similar assumptions by means of axioms, and recall these axioms during proofs. These assumptions often concern the correct behavior of the sensors, and sometimes are missing in the original description because implicitly assumed. This again shows that automatic support may help to uncover errors forcing the designer to precisely introduce every assumption. We also note that these assumptions are introduced by means of logical statements, similar to those given in [4], while a model checker would require us to express them in an operational way.

Another example we like to report here is the **Press Safety Property 2**: *The press does only close when no robot arm is positioned inside it.* In order to encode it in a concise form, we introduce two (derived) boolean functions: *PressIsClosing* and *ArmInPress,* defined as

```
PressIsClosing(s): bool = PressMot(PressPhase(s)) = up
```

i.e. the press is closing only when its motor is going up, and

```
ArmInPress(s): bool =
     Arm1Ext(s) > 0 and Angle(s) = Arm1ToPress or
     Arm2Ext(s) > 0 and Angle(s) = Arm2ToPress
```

Using these definitions the Press Safety property becomes:

```
PressSafety2: theorem
    PressIsClosing(s) => not ArmInPress(s)
```

To prove this property we model the angle of the robot arm by the monitored variable `Angle` encoded as function on the environment:

```
Angle :[ENV-> real]
```

Then we formalize all the assumptions about the movement of the robot: how the robot rotates, how the angle changes, and how the sensors communicate to the robot when to stop. We prove the Press Safety Property 2 using induction, case analysis (considering all the possible robot phases in the current and in the next state), recalling the assumptions about the robot movement, and applying the automatic decision procedures of PVS.

3.4 The Liveness Property

We have proved the liveness property as well, i.e. that the system never goes in deadlock. We have stated it as "*every component in the system will eventually change its state*". This statement is weaker than the property proved (by hand) in [4]. Börger and Mearelli establish also a performance valuation about the number of pieces that the cell is able to process cyclically. The proof of this performance property is under investigation. Our property is similar to the "progress agent property" stated in [4].

In order to prove liveness, we model and specify some assumptions about the environment (those called "Cell Assumption" in [4]). For example we assume that every object on the feed belt keeps moving until eventually it arrives at its destination and the monitored function `PieceInFeedBeltLightBarrier` becomes true:

```
FBAssumption: axiom
   FeedBeltPhase(s) = NormalRun =>
   exists(ns:(followingOrNow(s))):PieceInFeedBeltLightBarrier(ns)
```

`followingOrNow(s)` is the set of all states obtained repeatedly applying the function `next` to the state `s`, and of the state `s` itself.

We make similar assumptions for every state of every component of the cell, thus we assume that every component keeps moving till the monitored function of interest (which selected on the basis of the state and the component) eventually changes its value. Starting from these assumptions we prove, for example, the liveness of the FeedBelt:

```
FeedBeltProgress: lemma
   exists (ns: (following(s))):
       not FeedBeltPhase(ns) = FeedBeltPhase(s)
```

Using the same approach we are able to prove similar properties for every agent.

4 Related Work and Conclusions

Several attempts of applying both theorem provers and model checkers to ASM models have been performed. In [12] the KIV (Karlsruhe Interactive Verifier) system has been used to mechanically verify the proof of correctness of the Prolog to WAM transformation. PVS has been used in [7,8] to perform mechanical verification of the correctness of back-end rewrite system (BURS) specifications. A model checker approach is reported in [14] where correctness of the Production Cell specification of Börger and Mearelli has been proved through SMV. Recently, an interface from the ASM Workbench to the SMV model checking tool, based on an ASM-to-SMV transformation, has been presented in [6]. A direct comparison of our approach can be done with the work of Dold et al. using PVS and with the Winter's work using SMV.

Along our presentation, especially in remarks 1,2 and 3, we have discussed differences between Dold et al.'s approach and ours. We provide a more natural

translation of ASMs in PVS keeping the set of transition rules as a set of different rules, instead of forcing the user to define an equivalent transformation function in terms of one meta rule. We also provide the user with useful templates to guide his/her formalization of ASMs in PVS. These templates have also allowed us to provide a framework for a tool to automatically translate ASM specifications in PVS. This tool is under development and we plan to integrate it into the ASM-Workbench system. In addition we present proof schemes to encode and prove invariants and properties on traces.

Although the approach based on the model checker SMV allows properties verification in a completely automatic manner (unless the well known state explosion problem), the advantages of using our approach regard both the specification and the verification phase. PVS can easily manage specifications with infinite sets and an unbounded number of agents, has a powerful language (to represent functions, sets, lists and so on), has a strong type system, and can use the usual logical constructs (like universal and existential quantifications). Proof can be performed almost automatically in the simplest cases (as shown in the Production Cell case study). For more complex properties, in any case, our encoding can be used to check proofs done by hand or to support the user during the proof in an interactive way. In connection with the results presented in [14], we like to remark that the model checking approach can deal only with a finite set of agents and each agent having a finite number of possible states. This is the case of the Production Cell, under the assumption that continuous intervals (for example, the robot angle values) can be treated as finite sets of discrete values. This assumption was indeed used by Winter in [14], while it is not necessary in our approach, since we are able to deal with infinite sets (for example, we treat the robot angle as a real number). The correctness proof (with its results) of the Production Cell specification as it is shown in [14] has to be related to the added formalization of the environmental behavior. It is a mayor benefit of our approach that the assumptions regarding the interaction of the system and the environment can be formalized in a logical than in an operational way (i.e. in terms of transition rules) as required in [14].

Concluding, we like to stress our confidence that the proposed PVS encoding also works well for multi-agent ASMs with an unlimited number of agents, which are very complex to treat. It is not so hard to imagine how difficult can be performing mechanized proof verification of properties regarding interleaving computations of agents. The case study we present here is an example of a multi-agent system, but with a limited number of agents. However, the method has been successfully applied to analyze properties of crypto-protocols, where an unlimited number of agents run simultaneously. Security and authentication properties of the ASM specification presented in [1] have been proved in PVS using the technique of invariants. We have voluntarily left the presentation of these results out because they would require a specific treatment.

Acknowledgments. We kindly like to thank Egon Börger for his useful advice. We also thank anonymous referees for their helpful suggestions.

References

1. G. Bella and E. Riccobene. A Realistic Environment for Crypto-Protocol Analyses by ASMs. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
2. E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer, 1995.
3. E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
4. E. Börger and L. Mearelli. Integrating ASMs into the Software Development Life Cycle. *Journal of Universal Computer Science*, 3(5):603–665, 1997.
5. G. Del Castillo. The ASM Workbench: an Open and Extensible Tool Environment for Abstract State Machines. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
6. G. Del Castillo and K. Winter. Model Checking Support for the ASM High-Level Language. Technical Report TR-RI-99-209, Universität-GH Paderborn, June 1999.
7. A. Dold, T. Gaul, V. Vialard, and W. Zimmerman. ASM-Based Mechanized Verification of Compiler Back-Ends. In *Proceedings of the 28th Annual Conference of the German Society of Computer Science*. Technical Report, Magdeburg University, 1998.
8. Axel Dold. A formal representation of abstract state machines using pvs. Technical Report Verifix Report Ulm/6.2, Universitat Ulm, July 1998.
9. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
10. C. Heitmeyer. On the Need for Parctical Formal Methods. In *Proceedings of FTRTFT'98, 5th Intern. Symposium Real-Time Fault-Tolerant Systems*, volume 1486 of *LNCS*, pages 18–26. Springer, 1998.
11. C. Lewerentz and T. Linder, editors. *Formal Development of Reactive Systems. A Case Study "Production Cell"*. Number 891 in LNCS. Springer, 1995.
12. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.
13. N. Shankar, S. Owre, and J. Rushby. The PVS proof checker: A reference manual. Technical report, Computer Science Lab., SRI Intl., Menlo Park, CA, 1993.
14. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.

Model Checking Abstract State Machines and Beyond

Marc Spielmann

Mathematische Grundlagen der Informatik,
RWTH Aachen, D-52056 Aachen, Germany
spielmann@informatik.rwth-aachen.de

Abstract. We propose a systematic investigation of the (semi-) automatic verifiability of ASMs. As a first step, we put forward two verification problems concerning the correctness of ASMs and investigate the decidability and complexity of both problems.

1 Introduction

Abstract state machines (ASMs) [14,15,16] have become the formal foundation of a successful methodology for specification and verification of complex hardware and software systems. This is particularly witnessed by numerous publications using the ASM formalism for rigorous mathematical correctness proofs of large-scale applications. (See the recent bibliography [5] and the web site [24]. For an introduction to the ASM verification method the reader is referred to [7].) Interestingly, most of these contributions focus on *manual* verification, while the number of publications where all or part of the verification process is *automated* is rather small. (For exceptions see [25,26,12,6] and consult [24].) In a nutshell, computer-aided verification of ASMs, i.e., (semi-) automatic verification of dynamic systems expressed in terms of ASMs, has not yet been well developed. In view of the success of the ASM verification method in manual verification we think there is need for a *systematic investigation* of the (semi-) automatic verifiability of ASMs. The present paper can be viewed as an attempt to initiate such an investigation.

As a first step toward a systematic investigation of the verifiability of ASMs, we have to make precise what we actually mean by "verifying ASMs". In its full generality, the problem of verifying ASMs can be seen as a *decision problem* of the following kind:

> Given an ASM M (i.e., a formal description of some dynamic system) and a specification φ (i.e., a formal description of a desirable property of the system), decide whether M 'satisfies' φ.

One of our main goals in this paper is to identify decision problems of the above kind such that solving these problems coincides with proving properties of ASMs. We put forward two such problems, which we call the *model-checking problem* and the *verification problem* for ASMs.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 323–340, 2000.

The *model-checking problem* for ASMs, denoted MC, can be stated informally as follows:

> MC: Given an ASM M, a specification φ, and an input $\mathcal{I}$ appropriate for M, decide whether φ holds during all possible computations of M on $\mathcal{I}$.

Note that, in general, MC cannot be solved by means of testing, i.e., by simply running a given ASM M on a given input $\mathcal{I}$ and observing whether a given specification φ is satisfied. There are two reasons for this. Firstly, M may be non-deterministic or may access external functions or relations. For instance, if M chooses a natural number n in the first step and then proceeds depending on the choice of n, one would have to run M for each of the infinitely many possible choices for n. Secondly, even if M is deterministic and does not interact with its environment, it may not halt on $\mathcal{I}$.

The above formulation of MC immediately raises the question of a *specification language* for ASMs, i.e., a formal language suitable to express properties of ASMs. Since in the literature there is no consensus on the choice of such a language, we advocate here *first-order branching temporal logic* (FBTL) as specification language for ASMs. (FBTL is a straightforward extension of the well-known propositional branching-time logic CTL* by first-order reasoning [11]. For details see the next section.) Another open issue is the notion of *input*. The question is which type of input is suitable for ASMs and how does the initial state of an ASM on some input look like? Concerning this question we follow [2,4,13,3] and consider ASMs whose inputs are finite structures. To not impose unnecessary restrictions on the initial states of ASMs, we associate with every ASM M a function that maps every input $\mathcal{I}$ appropriate for M to an initial state of M on $\mathcal{I}$. In principle, this function can be any mapping from inputs to states.

An investigation of MC is motivated mainly by applications where it suffices to ensure correctness of an ASM for only a small number of inputs. As an example, consider an ASM M running on the notebook computer of a salesperson. M receives as input a database $\mathcal{D}$, say, the catalog of a supplier stored on a CD-ROM, and interacts with the salesperson via some external relations. The salesperson may inquire about the availability and prices of certain products or may store customer orders in a dynamic relation of M. Since in this scenario M runs on the given database $\mathcal{D}$ only, we can check whether M satisfies a specification φ by deciding whether $(M, \varphi, \mathcal{D})$ is a positive instance of MC.

Although a solution of MC is interesting for certain applications, there are many applications where correctness of an ASM must be guaranteed for *all* admissible inputs, of which there are often infinitely many. For instance, a compiler is usually supposed to correctly translate an input program of arbitrary length. In such cases one has to to check whether for a given ASM M and a given specification φ, $(M, \varphi, \mathcal{I})$ is a positive instance of MC for *every* admissible input $\mathcal{I}$. This problem, which we call the *verification problem* for ASMs and denote by VERIFY, can be stated informally as follows:

> VERIFY: Given an ASM M and a specification φ, decide whether for *every* input $\mathcal{I}$ appropriate for M, φ holds during all possible computations of M on $\mathcal{I}$.

Not surprisingly, both MC and VERIFY are—in their full generality—undecidable, as ASMs are computationally complete. (Indeed, every Turing machine can be regarded as a particularly simple ASM. It is an easy exercise to define a reduction of the halting problem for Turing machines to both problems.) Therefore, we investigate the decidability of MC and VERIFY with respect to *classes of restricted ASMs*. The idea is to formulate conditions on ASMs such that MC (resp. VERIFY) becomes decidable for ASMs that satisfy these conditions. Once decidability with respect to a class of ASMs is established, all ASMs in this class are automatically verifiable in the sense that there exists a procedure which checks whether a given ASM satisfies a given FBTL specification on a particular input (resp. on all inputs).

It is worth noticing that for unrestricted ASMs, MC and VERIFY collapse to the same problem. For example, in order to decide whether (M, φ) is a positive instance of VERIFY, it suffices to decide whether $(M', \varphi, \mathcal{I})$ is a positive instance of MC, where M' ignores $\mathcal{I}$, non-deterministically chooses some other input $\mathcal{I}'$ (of arbitrary size!), and then simulates M on $\mathcal{I}'$. However, MC and VERIFY do not necessarily coincide for restricted ASMs. Consider, for example, ASMs defined by means of a single rule of the form (`if` γ `then` *accept*), where the guard γ is a first-order sentence. Checking whether such an ASM accepts a particular input structure can be done in PSPACE, whereas verifying that it accepts all input structures is undecidable by Trakhtenbrot's theorem [10].

Although deciding MC appears to be easier than deciding VERIFY, it is a priori not clear whether a solution of VERIFY (for a class of ASMs) implies a solution of MC (for the same class of ASMs). The question is whether MC is a subproblem of VERIFY in the sense that there exists a reduction of MC to VERIFY with respect to fixed classes of ASMs. We give some formal evidence that this is indeed the case, thereby also providing a justification for the title of the paper.

Outline. In the next section, we recall first-order branching temporal logic which will serve here as specification language for ASMs. In Section 3, we formally define the model-checking problem for ASMs and present some results concerning its decidability and complexity (with respect to classes of restricted ASMs). In Section 4, we define the verification problem for ASMs, show that it subsumes the model checking problem, and study its decidability and complexity.

2 A Specification Language for Abstract State Machines

In order to investigate the automatic verifiability of ASMs we need to fix a formal language suitable to express properties of ASMs. We propose to specify properties of ASMs in terms of first-order branching temporal logic (FBTL). Formally, FBTL formulas express properties of *computation graphs* of ASMs. These graphs can be viewed as collections of runs of ASMs and are introduced next.

2.1 Computation Graphs

We assume the reader to be familiar with (sequential) ASMs [15,14]. To simplify the definition of computation graphs, we first introduce some additional terminology and notation.

ASM vocabularies. An *ASM vocabulary* Υ is a quadruple $(\Upsilon_{\text{in}}, \Upsilon_{\text{stat}}, \Upsilon_{\text{dyn}}, \Upsilon_{\text{ext}})$ of finite, pairwise disjoint vocabularies. The symbols in Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{ext} are called *input*, *static*, *dynamic*, and *external symbols*, respectively. Sometimes we also denote by Υ the (ordinary) vocabulary $\Upsilon_{\text{in}} \cup \Upsilon_{\text{stat}} \cup \Upsilon_{\text{dyn}} \cup \Upsilon_{\text{ext}}$. The intended meaning will be clear from the context. For technical reasons we shall always assume that Υ_{in} contains the constant symbol 0.

ASM programs. Let $\Upsilon = (\Upsilon_{\text{in}}, \Upsilon_{\text{stat}}, \Upsilon_{\text{dyn}}, \Upsilon_{\text{ext}})$ be an ASM vocabulary. An *ASM program* Π over Υ is a rule in the sense of [15], each of whose static (resp. dynamic, external) symbols occurs in $\Upsilon_{\text{in}} \cup \Upsilon_{\text{stat}}$ (resp. Υ_{dyn}, Υ_{ext}). Intuitively, the symbols in Υ_{stat} denote those relations and functions which are static and not included in the input of an ASM (e.g., arithmetical operations).

States. A *state* $\mathcal{S}$ over an ASM vocabulary Υ is a structure over the (ordinary) vocabulary Υ. We consider both finite and infinite states of ASMs.

Transitions. Let Π be an ASM program over Υ. The one-step semantics of Π defines a binary *transition relation* $T_{(\Upsilon,\Pi)}$ between states over Υ as usual. That is, $(\mathcal{S}, \mathcal{S}') \in T_{(\Upsilon,\Pi)}$ iff

1. $\mathcal{S}$ and $\mathcal{S}'$ are states over Υ, and
2. $\mathcal{S}'$ is the successor state (or sequel) of $\mathcal{S}$ with respect to Π in the sense of [15].

Remark 1. (a) If Π contains `choose`, `import`, or external symbols, then $T_{(\Upsilon,\Pi)}$ is in general non-deterministic, i.e., there may exist states $\mathcal{S}, \mathcal{S}', \mathcal{S}''$ with $(\mathcal{S}, \mathcal{S}')$, $(\mathcal{S}, \mathcal{S}'') \in T_{(\Upsilon,\Pi)}$ and $\mathcal{S}' \neq \mathcal{S}''$.

(b) If $\mathcal{S}$ is finite and Π contains `import`, then it may happen that Π attempts to import a new element, although the reserve of $\mathcal{S}$ is empty. One possible solution to this problem is to modify the semantics of `import` so that in states with empty reserve it always 'imports' the element denoted by 0. □

Inputs. An *input* $\mathcal{I}$ over an ASM vocabulary Υ is a finite structure over Υ_{in}.

Initialization mappings. An *initialization mapping initial* over an ASM vocabulary Υ is a function which maps every input $\mathcal{I}$ over Υ to a state $\mathcal{S}_\mathcal{I}$ over Υ satisfying the following three conditions:

1. The universe of $\mathcal{I}$ is a subset of the universe of $\mathcal{S}_\mathcal{I}$.
2. For every relation symbol $R \in \Upsilon_{\text{in}}$, every k-ary function symbol $f \in \Upsilon_{\text{in}}$, and every k-tuple $\bar{a}$ of elements in $\mathcal{S}_\mathcal{I}$,
$$R^{\mathcal{S}_\mathcal{I}} = R^\mathcal{I} \quad \text{and} \quad f^{\mathcal{S}_\mathcal{I}}(\bar{a}) = \begin{cases} f^\mathcal{I}(\bar{a}) & \text{if } \bar{a} \text{ consists of elements in } \mathcal{I} \\ 0^\mathcal{I} & \text{otherwise.} \end{cases}$$
3. For every input $\mathcal{I}'$ over Υ, if $\mathcal{I}'$ and $\mathcal{I}$ are isomorphic, so are $\mathcal{S}_{\mathcal{I}'}$ and $\mathcal{S}_\mathcal{I}$.

Abstract state machines. An *abstract state machine* M is a triple (Υ, *initial*, Π) consisting of

- an ASM vocabulary Υ,
- an initialization mapping *initial* over Υ, and
- an ASM program Π over Υ without free variables.

Υ_{in}, Υ_{stat}, Υ_{dyn}, and Υ_{ext} are called the *input*, *static*, *dynamic*, and *external vocabulary* of M, respectively. An *input* $\mathcal{I}$ appropriate for M is an input over Υ. M is *deterministic* if Π does not contain `choose`.

We can now define the notion of a computation graph of an ASM.

Definition 2. Let $M = (\Upsilon, \mathit{initial}, \Pi)$ be an ASM and $\mathcal{I}$ an input appropriate for M. The *computation graph* of M on $\mathcal{I}$, denoted $C_M(\mathcal{I})$, is a triple $(\mathit{States}, \mathit{Trans}, \mathcal{S}_0)$ where

- *States* is the set of those states over Υ whose universe is identical with the universe of $\mathit{initial}(\mathcal{I})$,
- $\mathit{Trans} \subseteq \mathit{States} \times \mathit{States}$ is the restriction of $T_{(\Upsilon,\Pi)}$ to *States*, and
- $\mathcal{S}_0 := \mathit{initial}(\mathcal{I})$.

A *run* of M on $\mathcal{I}$ is an infinite path in $C_M(\mathcal{I})$ starting at $\mathcal{S}_0$. □

Notice that for every ASM $M = (\Upsilon, \mathit{initial}, \Pi)$ and for every input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I})$ is finite iff $\mathit{initial}(\mathcal{I})$ is finite.

Example 3. In this and the next example (Example 5) we show that classical model checking, i.e., checking whether a given finite-state system is a model of a given temporal formula (see, e.g., [8]), is a special case of model checking ASMs. To this end, we display below a simple ASM M whose computation graphs are finite-state systems.

In classical model checking, finite-state systems are often represented as Kripke structures. A *Kripke structure* $\mathcal{K}$ is a tuple

$$(S, T, s_0, P_1, \ldots, P_k) \tag{1}$$

consisting of a finite set S of states, a binary transition relation $T \subseteq S \times S$, an initial state $s_0 \in S$, and a sequence $P_1, \ldots, P_k$ of subsets of S. (Sometimes, the sequence $P_1, \ldots, P_k$ is also given in the form of a labeling function $L : \{p_1, \ldots, p_k\} \to 2^S$ with $L(p_i) = P_i$ for each $i \in \{1, \ldots, k\}$.) Since it is customary in the context of model checking, we assume that every $s \in S$ has at least one T-successor, which may be s itself. $\mathcal{K}^- := (S, T, s_0)$ is called the *transition graph* of $\mathcal{K}$.

We define $M = (\Upsilon, \mathit{initial}, \Pi)$ so that for every Kripke structure $\mathcal{K}$, the computation graph of M on input $\mathcal{K}^-$ is isomorphic to $\mathcal{K}^-$. The vocabulary of M is defined as follows: $\Upsilon_{\text{in}} := \{T, 0\}$, where T is a binary relation symbol (which will be interpreted as the edge relation of the input transition graph), $\Upsilon_{\text{dyn}} := \{\mathit{pebble}\}$, where *pebble* is a nullary function symbol, $\Upsilon_{\text{stat}} := \varnothing$, and $\Upsilon_{\text{ext}} := \varnothing$. The program of M is:

```
program Π:
  choose x : T(pebble, x)
      pebble := x
```

Observe that the transition graph $\mathcal{K}^-$ of any Kripke structure $\mathcal{K}$ is a finite structure over Υ_{in}, and thus an input over Υ. The initialization mapping of M is defined by $\mathit{initial}(\mathcal{K}^-) := (\mathcal{K}^-, 0^{\mathcal{K}^-})$, where $(\mathcal{K}^-, 0^{\mathcal{K}^-})$ denotes the state over Υ in which T and 0 are interpreted as in $\mathcal{K}^-$, and the nullary dynamic function symbol *pebble* is interpreted as $0^{\mathcal{K}^-}$. One can now verify that $\mathcal{S} \mapsto \mathit{pebble}^{\mathcal{S}}$ is an isomorphism between $C_M(\mathcal{K}^-)$ and $\mathcal{K}^-$. □

2.2 First-Order Branching Temporal Logic

Recall that all runs of an ASM M on a particular input $\mathcal{I}$ are embedded in the computation graph of M on $\mathcal{I}$. Hence, it is reasonable to express a property of M (on any input) as a property of *all* computation graphs of M. To express properties of computation graphs we propose first-order branching temporal logic (FBTL), which is a combination of first-order logic and the well-known propositional branching temporal logic CTL* [8,11]. On the one hand, first-order logic allows us to reason about the states of an ASM. (Recall that the states of an ASM are first-order structures.) On the other hand, CTL* enables us to reason about the temporal and non-deterministic behavior of an ASM. For instance, one can express that 'good things eventually happen' or that 'there exists a good run' of an ASM. For the reader's convenience we recall the definition of FBTL here.

Definition 4. *State formulas* and *path formulas* of *first-order branching temporal logic* are defined by simultaneous induction:

(S1) Every atomic (first-order) formula is a state formula.
(S2) If φ is a path formula, then $\mathbf{E}\varphi$ is a state formula.
(P1) Every state formula is a path formula.
(P2) If φ and ψ are path formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are path formulas.
(SP1) If φ and ψ are state (resp. path) formulas, then $\varphi \vee \psi$ and $\neg\varphi$ are state (resp. path) formulas.
(SP2) If x is a variable and φ a state (resp. path) formula, then $\exists x\varphi$ is a state (resp. path) formula.

The *free* and *bound variables* of state and path formulas are defined in the obvious way. FBTL denotes the set of state formulas. □

Semantics of FBTL formulas. Intuitively, a state formula of the form $\mathbf{E}\varphi$ expresses that there (E)xists an infinite path (presumably a run of an ASM) such that the path formula φ holds along this path. The intuitive meaning of path formulas of the form $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ is as follows:

– $\mathbf{X}\varphi$: φ holds in the ne(X)t state.

- $\varphi \mathbf{U} \psi$: ψ holds eventually and φ holds (U)ntil then.
- $\varphi \mathbf{B} \psi$: either ψ holds always or φ holds (B)efore ψ fails.

We formally define the semantics of FBTL formulas only with respect to computation graphs of ASMs. Let Υ be an ASM vocabulary, φ a state formula over Υ, ψ a path formula over Υ, and $C = (\mathit{States}, \mathit{Trans}, \mathcal{S}_0)$ a computation graph of an ASM of vocabulary Υ. W.l.o.g., we may assume that $\mathrm{free}(\varphi) = \mathrm{free}(\psi) = \{\bar{x}\}$. Let $\rho = (\mathcal{S}'_i)_{i \in \omega}$ be an infinite path in C (not necessarily starting at the initial state $\mathcal{S}_0$ of C). For any $j \in \omega$, let $\rho|j$ denote the infinite path $(\mathcal{S}'_{i+j})_{i \in \omega}$, i.e., the suffix $\mathcal{S}'_j, \mathcal{S}'_{j+1}, \ldots$ of ρ.

Simultaneously for every state $\mathcal{S}$ in C, every infinite paths ρ in C, and all interpretations $\bar{a}$ of the variables $\bar{x}$ (chosen from the universe of $\mathcal{S}_0$) define the two *satisfactory relations* $(C, \mathcal{S}, \bar{a}) \models \varphi$ and $(C, \rho, \bar{a}) \models \psi$ by induction on the construction of φ and ψ:

(S1)	$(C, \mathcal{S}, \bar{a}) \models \varphi$	$:\Leftrightarrow$	$\mathcal{S} \models \varphi[\bar{a}]$, where φ is an atomic formula
(S2)	$(C, \mathcal{S}, \bar{a}) \models \mathbf{E}\varphi$	$:\Leftrightarrow$	there is an infinite path ρ' in C starting at $\mathcal{S}$ such that $(C, \rho', \bar{a}) \models \varphi$
(P1)	$(C, \rho, \bar{a}) \models \varphi$	$:\Leftrightarrow$	$(C, \mathcal{S}'_0, \bar{a}) \models \varphi$, where $\mathcal{S}'_0$ is the first state of ρ
(P2)	$(C, \rho, \bar{a}) \models \mathbf{X}\varphi$	$:\Leftrightarrow$	$(C, \rho\|1, \bar{a}) \models \varphi$
	$(C, \rho, \bar{a}) \models \varphi \mathbf{U} \psi$	$:\Leftrightarrow$	there exists $i \in \omega$ such that $(C, \rho\|i, \bar{a}) \models \psi$ and for all $j < i$, $(C, \rho\|j, \bar{a}) \models \varphi$
	$(C, \rho, \bar{a}) \models \varphi \mathbf{B} \psi$	$:\Leftrightarrow$	for every $i \in \omega$, if $(C, \rho\|i, \bar{a}) \models \neg\psi$, then there exists $j < i$ with $(C, \rho\|j, \bar{a}) \models \varphi$

The semantics of formulas derived by means of rule **(SP1)** is standard. It remains to declare the semantics of formulas derived by means of rule **(SP2)**. Below, σ stands for either a state $\mathcal{S}$ or an infinite path ρ in C, depending on whether φ is a state or a path formula.

(SP2)	$(C, \sigma, \bar{a}) \models \exists y \varphi(\bar{x}, y)$	$:\Leftrightarrow$	there is an element b in the universe of $\mathcal{S}_0$ such that $(C, \sigma, \bar{a}, b) \models \varphi(\bar{x}, y)$

For every FBTL sentence φ over Υ, let

$$C \models \varphi \quad :\Leftrightarrow \quad (C, \mathcal{S}_0) \models \varphi.$$

The following abbreviations are customary and will be used frequently:

- $\mathbf{A}\varphi := \neg\mathbf{E}\neg\varphi$ (φ holds along every path).
- $\mathbf{F}\varphi := \mathit{true}\,\mathbf{U}\varphi$ (φ holds eventually).
- $\mathbf{G}\varphi := \mathit{false}\,\mathbf{B}\varphi$ (φ holds always).

Note that $\varphi \mathbf{B} \psi \equiv \neg(\neg\varphi \mathbf{U} \neg\psi)$.

Example 5. Recall the ASM M in Example 3. We observe that classical model checking is a special case of model checking M. Let $\Upsilon' = \{P_1, \ldots, P_k\}$ be a vocabulary containing only set symbols, and *pebble* a nullary function symbol (as in the vocabulary of M). For every CTL* formula φ over Υ', obtain the

FBTL sentence φ' over $\Upsilon' \cup \{pebble\}$ by replacing in φ every occurrence of P_i with the atomic formula $P_i(pebble)$. Let $M_{\Upsilon'}$ be defined as M, except that the input vocabulary of $M_{\Upsilon'}$ is $\{T, 0\} \cup \Upsilon'$ and the initialization mapping of $M_{\Upsilon'}$ maps every Kripke structure $\mathcal{K}$ of form (1) to the state $(\mathcal{K}, 0^{\mathcal{K}})$. Now verify that for every CTL* formula φ over Υ' and every Kripke structure $\mathcal{K}$ of form (1),

$$\mathcal{K} \models \varphi \quad \Leftrightarrow \quad C_{M_{\Upsilon'}}(\mathcal{K}) \models \varphi'.$$

In other words, we can check whether $\mathcal{K}$ is a model of φ by deciding whether $(M_{\Upsilon'}, \varphi', \mathcal{K})$ is a positive instance of the model-checking problem for ASMs. In fact, in the next section we are going to define the model-checking problem for ASMs such that $\mathcal{K} \models \varphi$ iff $(M_{\Upsilon'}, \varphi', \mathcal{K}) \in$ MC. □

Examples of FBTL specifications motivated by concrete applications can be found in [22,23].

3 Model Checking Abstract State Machines

Recall the informal formulation of the model-checking problem for ASMs in the introduction. If the reader agrees that FBTL (or some suitable extension of this logic) is an appropriate specification language for ASMs, then the model-checking problem for ASMs can be rephrased as follows:

MC$'$: Given an ASM M, a FBTL sentence φ over the vocabulary of M, and an input $\mathcal{I}$ appropriate for M, decide whether $C_M(\mathcal{I}) \models \varphi$.

Notice, however, that this problem is still **not** a computational problem. The difficulty here is that the initialization mapping of an ASM M may not be finitely representable, in which case M cannot serve as input to a computational device.

Next, we further restrict the above problem so that it becomes a computational problem. Then, in the second part of this section, we present some results concerning the decidability and complexity of the obtained computational problem.

3.1 The Model-Checking Problem

The problem of representing the initialization mappings of ASMs in finite terms disappears if we concentrate on ASMs uniformly initialized in the following sense.

Definition 6. A class C of ASMs is *uniformly initialized* if, whenever M and M' are two ASMs in C of the same vocabulary, then M and M' have that same initialization mapping. □

With every uniformly initialized class C of ASMs and every ASM vocabulary Υ, we can associate an initialization mapping $initial^C_\Upsilon$, such that $initial^C_\Upsilon$ is the initialization mapping of every $M \in C$ of vocabulary Υ. Observe that, in the context of a uniformly initialized class C of ASMs, every $M \in C$ is uniquely determined by its vocabulary and program. More precisely, if Υ is the vocabulary and Π the program of M, then necessarily $M = (\Upsilon, initial^C_\Upsilon, \Pi)$. This motivates the following definition.

Definition 7. The *standard representation* of an ASM $(\Upsilon, \mathit{initial}, \Pi)$ is given by the pair (Υ, Π). □

We are now in the position to define the model-checking problem for ASMs as a computational problem. Let C be a uniformly initialized class of ASMs and F a fragment of FBTL. The *model checking problem for C-ASMs and F-specifications* is the following decision problem:

MC(C, F): Given the standard representation of an ASM $M \in C$, a sentence $\varphi \in F$ over the vocabulary of M, and an input $\mathcal{I}$ appropriate for M, decide whether $C_M(\mathcal{I}) \models \varphi$.

Remark 8. Note the subtlety in the above definition of MC, namely that by viewing an ASM $M \in C$ as a finitely representable pair (Υ, Π) the problem of actually representing the initialization mapping of M has not been solved, but rather has been made part of the model-checking problem itself. □

Example 9. (a) For every vocabulary $\Upsilon' = \{P_1, \ldots, P_k\}$ containing only set symbols, and every CTL* formula φ over Υ', let $M_{\Upsilon'}$ and φ' be defined as in Example 5. Let C denote the class of all $M_{\Upsilon'}$, and set $F = \{\varphi' : \varphi \in \text{CTL}^*\}$. C is uniformly initialized and we have $(M_{\Upsilon'}, \varphi', \mathcal{K}) \in \text{MC}(C, F)$ iff $\mathcal{K} \models \varphi$. That is, MC$(C, F)$ coincides with the model checking problem for CTL*.

(b) One can define a class C of ASMs and a fragment F of FBTL such that MC(C, F) coincides with the *symbolic* model checking problem for CTL*. We sketch the idea. Consider a Kripke structure $\mathcal{K}$ of form (1) and a CTL* formula φ over $\{P_1, \ldots, P_k\}$. Obtain the FBTL formula φ' by replacing every occurrence of P_i with the boolean symbol p_i. (A *boolean symbol* is a nullary relation symbol.) $\mathcal{K}$ can be represented by means of boolean formulas $\gamma_T(\bar{x}, \bar{y}), \gamma_{P_1}(\bar{x}), \ldots, \gamma_{P_k}(\bar{x})$ with $\bar{x}$ and $\bar{y}$ two n-tuples of boolean variables, $\text{free}(\gamma_T) = \{\bar{x}, \bar{y}\}$, and for each P_i, $\text{free}(\gamma_{P_i}) = \{\bar{x}\}$. For simplicity, let us assume that for each P_i, $\gamma_{P_i}(\overline{\mathit{false}})$ evaluates to *false*. There is an ASM $M_{\mathcal{K}}$ with input vocabulary $\{0\}$, dynamic vocabulary $\{\bar{x}, \bar{p}\}$, and

```
program Π:
  choose y_1, ..., y_n ∈ {true, false} : γ_T(x̄, ȳ)
     x̄ := ȳ
     if γ_{P_1}(ȳ) then p_1 := true else p_1 := false
     ...
     if γ_{P_k}(ȳ) then p_k := true else p_k := false
```

such that for any input $\mathcal{I}$ appropriate for $M_{\mathcal{K}}$, $\mathcal{K} \models \varphi \Leftrightarrow C_{M_{\mathcal{K}}}(\mathcal{I}) \models \varphi'$. □

In the remainder of this section we investigate the decidability and complexity of MC(C, F).

3.2 Decidability and Complexity

We first observe that the model-checking problem is decidable for uniformly initialized ASMs whose initial states are finite and computable from the inputs.

Lemma 10. *Let C be a uniformly initialized class of ASMs such that there exists an algorithm A_C as follows. For every ASM vocabulary Υ and every input $\mathcal{I}$ over Υ, A_C on input $(\Upsilon, \mathcal{I})$ outputs $initial_\Upsilon^C(\mathcal{I})$ (where $initial_\Upsilon^C$ is defined as below Definition 6). Then* $\textsc{mc}(C, \mathrm{FBTL})$ *is decidable.*

Proof. Given an instance $(M, \varphi, \mathcal{I})$ of $\textsc{mc}(C, \mathrm{FBTL})$, where the vocabulary of M is, say, Υ, we can run A_C on $(\Upsilon, \mathcal{I})$ to obtain the initial state of M on $\mathcal{I}$. Since this state is finite, so is $C_M(\mathcal{I})$. Hence, we can construct $C_M(\mathcal{I})$ explicitly and check whether $C_M(\mathcal{I}) \models \varphi$ holds. □

Applications of this lemma can be found in [25,9,26]. Note however that the decision procedure outlined in the proof of the lemma is often too expensive to be useful in practice, even if the complexity of the algorithm A_C is low. For example, the number of reachable states of an ASM M employing a non-nullary dynamic relation in general grows exponentially in the size of the initial state. Consequently, the space required for constructing a computation graph of M explicitly can be exponential in the size of the input, independent of A_C.

Next, we present a class C of ASMs and a fragment F of FBTL for which a useful restriction of $\textsc{mc}(C, F)$ is in PSPACE.

Definition 11. An ASM $M = (\Upsilon, initial, \Pi)$ is *finitely initialized* if for every input $\mathcal{I}$ appropriate for M,

1. the universe of $initial(\mathcal{I})$ is that of $\mathcal{I}$, and
2. for every relation symbol $R \in \Upsilon - \Upsilon_{\mathrm{in}}$, every k-ary function symbol $f \in \Upsilon - \Upsilon_{\mathrm{in}}$, and every k-tuple $\bar{a}$ of elements in $initial(\mathcal{I})$,

$$R^{initial(\mathcal{I})} = \varnothing \quad \text{and} \quad f^{initial(\mathcal{I})}(\bar{a}) = 0.$$

□

Observe that the initialization mapping of every finitely initialized ASM M is uniquely determined by the vocabulary of M. In particular, every class of finitely initialized ASMs is uniformly initialized and for every such class C there exists an algorithm A_C as in Lemma 10.

Remark 12. (a) On the first glance, the second condition in Definition 11 may seem to render finitely initialized ASMs useless for practical purposes, because in applications the initial interpretations of the symbols in $\Upsilon - \Upsilon_{\mathrm{in}}$ must often satisfy certain conditions. In particular, it may not be desirable to have all symbols in $\Upsilon - \Upsilon_{\mathrm{in}}$ be initialized as in the second condition of the definition. But note that the initial interpretations of these symbols can be seen as part of the input of an ASM. For example, consider a finitely initialized ASM M whose nullary dynamic function *pebble* should assume, instead of 0, the value 42 in the initial state. This can be achieved by modifying M as follows. Let Π be the program of M, *pebbleInitial* a new constant symbol, and *running* a new boolean symbol. Replace Π with the following program:

```
program Π′:
  if ¬running then
     pebble := pebbleInitial
     running := true
  if running then
     Π
```

Now, while model checking M, consider only input structures in which the input symbol *pebbleInitial* is interpreted as 42.

(b) Suppose that the second condition in Definition 11 is relaxed as follows: there is a distinguished symbol $S \in \Upsilon - \Upsilon_{\text{in}}$ whose initial interpretation depends on Υ but is not restricted otherwise. We display a class C of simple ASMs finitely initialized in the relaxed sense and a simple fragment F of FBTL such that $\text{MC}(C, F)$ is undecidable.

W.l.o.g., we may assume that S is a boolean symbol and that every ASM vocabulary contains S. Choose some undecidable problem

$$P \subseteq \{\Upsilon : \Upsilon \text{ is an ASM vocabulary}\}.$$

(For instance, let TM_Υ be the Turing machine whose encoding—in some fixed standard binary encoding—equals the binary representation of the number of symbols in Υ. Then $P := \{\Upsilon : TM_\Upsilon \text{ halts on the empty word}\}$ is undecidable, as a reduction of the halting problem for Turing machines shows.) For every ASM vocabulary Υ, let $M_\Upsilon := (\texttt{skip}, \mathit{initial}_\Upsilon, \Upsilon)$ be a finitely initialized ASM, except that now for every input $\mathcal{I}$ appropriate for M, $\mathit{initial}_\Upsilon(\mathcal{I}) \models S$ iff $\Upsilon \in P$. Let C denote the class of all M_Υ, and set $F = \{S\}$. For any input $\mathcal{I}$ appropriate for M, $\Upsilon \mapsto (M_\Upsilon, S, \mathcal{I})$ is a reduction of P to $\text{MC}(C, F)$. This implies that $\text{MC}(C, F)$ is undecidable. □

In favor of a succinct formulation of the next theorem we introduce some additional notation.

Definition 13. Let T denote the closure of the set of first-order formulas under the rules for negation and disjunction, and the following rule:

(T) If φ and ψ are formulas, then $\mathbf{X}\varphi$, $\varphi\mathbf{U}\psi$, and $\varphi\mathbf{B}\psi$ are formulas.

The *universal closure of* T, denoted UT, is the set of formulas of the form $\mathbf{A}\forall\bar{x}\varphi$ with $\varphi \in \text{T}$ and $\text{free}(\varphi) \subseteq \{\bar{x}\}$, where $\text{free}(\varphi)$ is defined in the obvious way. □

We define a *restriction* of the model-checking problem for ASMs. An investigation of this restriction is motivated by the observation that the arities of relations and functions used in practice tend to be rather small. Indeed, for practical purposes it often suffices to solve MC for ASMs whose vocabulary contains only symbols of arity $\leq m$, for some a priori fixed natural number m. Let $\text{MC}_m(C, F)$ denote the restriction of $\text{MC}(C, F)$ to instances where only symbols of arity $\leq m$ occur.

The following theorem generalizes a result in [22]. Note that UT can be viewed as a fragment of FBTL.

Theorem 14. *Let C be the class of finitely initialized ASMs whose program does not contain* `import` *or* `choose`*. For any natural number m, $\text{MC}_m(C, \text{UT})$ is* PSPACE*-complete.*

We already encounter PSPACE-hardness of $\text{MC}_m(C, \text{UT})$ if $m = 0$ and C is the class of finitely initialized ASMs of the form $(\texttt{skip}, \mathit{initial}_\Upsilon, \Upsilon)$. This follows by a reduction of the PSPACE-complete model-checking problem for propositional temporal logic [20,11].

As already pointed out in the introduction, a decision procedure for $\text{MC}(C, F)$ can be useful for the verification of ASMs that are supposed to run correctly on a small number of inputs only. However, for applications where correctness of an ASM has to be ensured for a large number of inputs—or even infinitely many inputs—a solution of $\text{MC}(C, F)$ does not suffice. For such applications we have to solve the model-checking problem for *all* admissible inputs. This will be our main concern in the next section.

4 Verifying Abstract State Machines

We study the problem of model checking ASMs against FBTL specifications for *all* inputs. After formally defining this problem, which we call the *verification problem* for ASMs, we provide some evidence that it subsumes the model-checking problem for ASMs and investigate its decidability and complexity.

4.1 The Verification Problem

Let C be a uniformly initialized class of ASMs and F a fragment of FBTL. *Verifying C-ASMs against F-specifications* means solving the following decision problem:

VERIFY(C, F): Given the standard representation of an ASM $M \in C$ and a sentence $\varphi \in F$ over the vocabulary of M, decide whether for every input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I}) \models \varphi$.

Remark 15. One may object that the above formulation of VERIFY does not adequately reflect real-life verification of ASMs, as in applications one is often interested in verifying an ASM only for inputs that satisfy certain conditions. Note however that these conditions can be viewed as part of a specification. For example, suppose that the FBTL sentence ψ describes all admissible inputs of an ASM M. That is, an input $\mathcal{I}$ is considered to be admissible for M iff $\mathcal{I} \models \psi$. Then $(M, \psi \to \varphi)$ is a positive instance of VERIFY iff for every admissible input $\mathcal{I}$ appropriate for M, $C_M(\mathcal{I}) \models \varphi$. □

Not surprising, $\text{VERIFY}(C, F)$ is in general undecidable, even for classes C of simple ASMs and simple fragments F of FBTL. For instance, recall the reduction of the undecidable problem P to $\text{MC}(C, F)$ in Remark 12 (b) and observe that the same reduction (with $\mathcal{I}$ now removed from the image of every Υ) also reduces P to $\text{VERIFY}(C, F)$.

Proviso 16. *In the remainder of this section we restrict ourselves to finitely initialized ASMs (see Definition 11).* □

4.2 Model Checking vs Verification

We show that MC(C, F) is polynomial-time reducible to VERIFY(C, F) if C and F satisfy certain closure conditions. In favor of a concise presentation of the result we introduce a restricted type of first-order quantification, which we call *witness-bounded quantification*. In the following, FO denotes first-order logic (with equality).

Witness-bounded quantification. A *witness set* W is a finite set of variables and constant symbols. For a witness set W and a variable x not in W, we write $(x \in W)$ instead of $(\bigvee_{\nu \in W} x = \nu)$. Intuitively, $(x \in W)$ holds iff the interpretation of x matches the interpretation of some symbol in W.

Definition 17. The *witness-bound fragment* of FO, denoted FO^{W}, is obtained from FO by replacing the rule for (unrestricted) quantification with the following rule for *witness-bounded quantification*:

(WBQ) If W is a witness set, x a variable not in W, and φ a formula, then $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are formulas.

The free and bound variables of FO^{W} formulas are defined as usual. In particular, x occurs bound in $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$, while all variables in the witness set W occur free. □

We view FO^{W} as a fragment of FO where formulas of the form $(\exists x \in W)\varphi$ and $(\forall x \in W)\varphi$ are mere abbreviations for $\exists x(x \in W \wedge \varphi)$ and $\forall x(x \in W \to \varphi)$, respectively.

Reduction of MC to VERIFY. Proceeding toward a reduction of MC to VERIFY, we next define witness-bounded versions of ASMs and FBTL formulas. Let W be a witness set containing only constant symbols. For every FBTL formula φ, obtain φ^W from φ by replacing every first-order quantifier with the corresponding W-bounded quantifier (e.g., replace $\exists x$ with $(\exists x \in W)$, and $\forall x$ with $(\forall x \in W)$). For every ASM $M = (\Upsilon, \mathit{initial}, \Pi)$ with $W \subseteq \Upsilon_{\mathrm{in}}$, let $M^W := (\Upsilon, \mathit{initial}, \Pi^W)$ where Π^W is obtain by induction on the construction of Π:

- If Π is `skip` or an update, then $\Pi^W := \Pi$.
- If $\Pi = (\texttt{if}\ \varphi\ \texttt{then}\ \Pi_0)$, then $\Pi^W := (\texttt{if}\ \varphi^W\ \texttt{then}\ \Pi_0^W)$.
- If $\Pi = (\Pi_0 || \Pi_1)$, then $\Pi^W := (\Pi_0^W || \Pi_1^W)$.
- If $\Pi = (\texttt{do-forall}\ x : \varphi, \Pi_0)$, then $\Pi^W := (\texttt{do-forall}\ x \in W : \varphi^W, \Pi_0^W)$.

If Π is a `choose` or `import` rule, proceed similar to the last case. For a class C of ASMs and a fragment F of FBTL, let C^{W} and F^{W} be defined by:

$$\begin{aligned} C^{\mathrm{W}} &:= \{M^W : M = (\Upsilon, \mathit{initial}, \Pi) \in C, \\ &\qquad W \subseteq \Upsilon_{\mathrm{in}} \text{ is a set of constant symbols}\} \\ F^{\mathrm{W}} &:= \{\varphi^W : \varphi \in F,\ W \text{ is a set of constant symbols}\}. \end{aligned}$$

Lemma 18. *Let C_0 be the class of finitely initialized ASMs whose program does not contain* `import` *and whose input (resp. external) vocabulary does not contain function symbols of arity ≥ 1 (resp. ≥ 0). Furthermore, let F be a fragment of* FBTL *containing for every boolean symbol b the formula* $\mathbf{EX}b$*. For any class $C' \supseteq C_0^{\mathrm{W}}$ and any fragment $F' \supseteq F^{\mathrm{W}}$ closed under the boolean operators,* $\textsc{mc}(C_0, F)$ *is polynomial-time reducible to* $\textsc{verify}(C', F')$.

Proof. (Sketch.) Consider an instance $(M, \varphi, \mathcal{I})$ of $\textsc{mc}(C, F)$ with $M = (\Upsilon, \mathit{initial}_\Upsilon, \Pi)$. W.l.o.g., we may assume that every element in $\mathcal{I}$ is denoted by some constant symbol in Υ_{in}. (If this is not the case, enrich Υ_{in} with new constant symbols and add interpretations of the new symbols to $\mathcal{I}$. This modification of $\mathcal{I}$ is clearly polynomial-time computable.) We construct an instance (M', φ') of $\textsc{verify}(C^{\mathrm{W}}, F')$ which is positive iff $(M, \varphi, \mathcal{I})$ is a positive instance of $\textsc{mc}(C, F)$.

Let $\chi_\mathcal{I} := \bigwedge_{\mathcal{I} \models \gamma} \gamma$ where γ ranges in the set of atomic and negated atomic sentences over Υ_{in}. $\chi_\mathcal{I}$ is a quantifier-free FO sentence over Υ_{in} and can be constructed from $\mathcal{I}$ in polynomial time. For every finite structure $\mathcal{I}'$ over Υ_{in} we have $\mathcal{I}' \models \chi_\mathcal{I}$ iff there exists a substructure of $\mathcal{I}'$ isomorphic to $\mathcal{I}$.

To the definition of M' and φ'. Let W be the set of constant symbols in Υ_{in}, and *error* a boolean symbol not in Υ. Set $\varphi' = (\neg \mathbf{EX} \mathit{error} \rightarrow \varphi^W)$ and $M' = (\Upsilon, \mathit{initial}_\Upsilon, \Pi')$ with $\Pi' := (\Pi^W \mid\mid \texttt{if } \neg\chi_\mathcal{I} \texttt{ then } \mathit{error})$. (M', φ') can be obtained from $(M, \varphi, \mathcal{I})$ in polynomial time and is an instance of $\textsc{verify}(C^{\mathrm{W}}, F')$. □

Let C_0 be as in the above lemma. The proof of the lemma shows that, if $C \subseteq C_0$ and $F \subseteq$ FBTL have reasonable closure properties and $C^{\mathrm{W}} \subseteq C$ and $F^{\mathrm{W}} \subseteq F$, then $\textsc{mc}(C, F)$ is polynomial-time reducible to $\textsc{verify}(C, F)$.

4.3 Decidability and Complexity

The next proposition provides a sufficient condition for the decidability of the verification problem. Let us first recall two decision problems from logic.

Finite satisfiability and finite validity. Let L be a logic, Υ a vocabulary, and φ an L-sentence over Υ. φ is called *finitely satisfiable* if there exists a finite structure $\mathcal{A}$ over Υ with $\mathcal{A} \models \varphi$. φ is called *finitely valid* if for every finite structure $\mathcal{A}$ over Υ, $\mathcal{A} \models \varphi$. By $\textsc{fin-sat}(L)$ (resp. $\textsc{fin-val}(L)$) we denote the problem of deciding finite satisfiability (resp. finite validity) of a given L-sentence.

Proposition 19. *(a) Let C be a class of ASMs and F a fragment of* FBTL*. If there exists a logic L satisfying the following two conditions, then* $\textsc{verify}(C, F)$ *is decidable.*

1. $\textsc{fin-val}(L)$ *is decidable.*
2. *There exists a computable function which maps every instance (M, φ) of* $\textsc{verify}(C, F)$ *to an L-sentence $\chi_{M,\varphi}$ over the input vocabulary of M, such that for every input $\mathcal{I}$ appropriate for M,*

$$C_M(\mathcal{I}) \models \varphi \quad \Leftrightarrow \quad \mathcal{I} \models \chi_{M,\varphi}.$$

(b) Let $\neg F$ denote the set of negated F-sentences. VERIFY$(C, \neg F)$ *is decidable if in the first condition instead of* FIN-VAL(L), FIN-SAT(L) *is decidable.*

Proof. To assertion (a). The second condition immediately implies that $(M, \varphi) \mapsto \chi_{M,\varphi}$ is a reduction of VERIFY(C, F) to FIN-VAL(L). The latter problem is decidable by assumption. To assertion (b). Verify the following chain of equivalences: $(M, \neg\varphi) \in$ VERIFY$(C, \neg F)$ iff $C_M(\mathcal{I}) \models \neg\varphi$ for every $\mathcal{I}$ (appropriate for M) iff $C_M(\mathcal{I}) \not\models \varphi$ for every $\mathcal{I}$ iff $\mathcal{I} \not\models \chi_{M,\varphi}$ for every $\mathcal{I}$ iff $\chi_{M,\varphi} \notin$ FIN-SAT(L). Hence, $(M, \neg\varphi) \mapsto \chi_{M,\varphi}$ is a reduction of VERIFY$(C, \neg F)$ to the complement of FIN-SAT(L). Decidability of FIN-SAT(L) then implies assertion (b). □

Of course, the main challenge in applying Proposition 19 is to find a logic L that satisfies the two conditions in the proposition. As possible candidates for L we propose the following logics:

- existential transitive closure logic (E+TC),
- existential least fixed-point logic (E+LFP) [1,10], and
- SO($\exists$), i.e., the set of second-order formulas in prenex normal form, whose quantifier prefix is an arbitrary string of SO quantifiers followed be a sting of existential FO quantifiers [19].

Both FIN-VAL and FIN-SAT are decidable for each of these logics if one restricts attention to formulas over relational vocabularies [18,19].

In the remainder of this section we recall some results concerning the decidability of VERIFY from [21]. The main positive result there was obtained by means of an application of Proposition 19.

Definition 20. Let Υ be an ASM vocabulary where Υ_{dyn} and Υ_{ext} contain only nullary symbols. *Sequential nullary programs* over Υ are defined inductively:

- **Updates:** For every relation symbol $b \in \Upsilon_{\text{dyn}}$, every function symbol $v \in \Upsilon_{\text{dyn}}$, and every term t over Υ, each of the following is a sequential nullary program: $b := \mathit{true}$, $b := \mathit{false}$, $v := t$.
- **Conditionals:** If Π is a sequential nullary program and φ a witness-bounded formula over Υ, then (`if` φ `then` Π) is a sequential nullary program.
- **Parallel composition:** If Π_0 and Π_1 are sequential nullary programs, then $(\Pi_0 || \Pi_1)$ is a sequential nullary program.
- **Non-deterministic choice:** If Π is a sequential nullary program, $\bar{x}$ a tuple of pairwise distinct variables, and φ a witness-bounded formula over Υ such that $\exists \bar{x} \varphi$ is finitely valid, then (`choose` $\bar{x} : \varphi$, Π) is a sequential nullary program.

The *free* and *bound variables* of a sequential nullary program are defined in the obvious way.

A *sequential nullary ASM* is a finitely initialized ASM whose program is a sequential nullary program without free variables. SN-ASM$_{\text{rel}}$ (resp. SN-ASM$_{\text{fct}}$) denotes the class of sequential nullary ASMs whose input vocabulary contains only relation and constant symbols (resp. contains at least two non-nullary symbols, one of which is a function symbol). □

The next definition introduces two fragments of FBTL, denoted ETE and UTU. Informally speaking, the formulas in ETE (resp. UTU) are built from atomic formulas by means of disjunction, conjunction, existential (resp. universal) quantification (applicable only to state formulas), the temporal operators **X**, **U**, and **B**, and the path quantifier **E** (resp. **A**).

Definition 21. ETE denotes the set of FBTL formulas derivable by means of rules **(S1)**, **(S2)**, **(P1)**, **(P2)** in Definition 4, and the following two formula-formation rules:

(SP1)' If φ and ψ are state (resp. path) formulas, then $\varphi \vee \psi$ and $\varphi \wedge \psi$ are state (resp. path) formulas.
(SP2)' If x is a variable and φ a state formula, then $\exists x \varphi$ is a state formula.

UTU denotes the set of negated ETE formulas. □

Let $\textsc{verify}_m(C, F)$ denote the restriction of $\textsc{verify}(C, F)$ to instances where only symbols of arity $\leq m$ occur. Again, solving $\textsc{verify}_m(C, F)$ instead of $\textsc{verify}(C, F)$ should be no serious obstacle for practical purposes.

Theorem 22 ([21]). *For any natural number m, the problem*

$$\textsc{verify}_m(\text{SN-ASM}_{\text{rel}}, \text{ETE} \cup \text{UTU})$$

is Pspace*-complete. In other words, verifying sequential nullary ASMs with relational input against* (ETE ∪ UTU)*-specifications is a* Pspace*-complete problem, given that the maximal arity of the employed input relations is a priori bounded.*

The proof of the theorem closely follows a construction due to Immerman and Vardi that was first presented in [17] as a translation of CTL* into (FO+TC). Using this construction one can show that for $C = \text{SN-ASM}_{\text{rel}}$, $F = \text{ETE}$, and $L = (\text{E+TC})$ the second condition in Proposition 19 is satisfied. The containment assertion of the theorem is then implied by the following two observations:

1. Finite validity and finite satisfiability of (E+TC) sentences over relational vocabularies is *decidable in* Pspace if one imposes an upper bound m on the arities of the occurring relation symbols [21,23].
2. There exists a *polynomial-time* computable function as in the second condition of Proposition 19.

$\textsc{verify}_m(C, F)$ is already Pspace-hard if $m = 0$, F contains $\mathbf{EX}\mathit{accept}$, and C includes all ASMs in $\text{SN-ASM}_{\text{rel}}$ whose external vocabulary is empty and whose dynamic vocabulary contains only boolean symbols. This follows by a reduction of the Pspace-complete satisfiability problem for quantified boolean formulas.

Unfortunately, Theorem 22 does not hold for sequential nullary ASMs whose inputs contain functions, as the following theorem shows. For the sake of brevity we write $\textsc{liveness}(C)$ instead of $\textsc{verify}(C, \{\mathbf{EF}\mathit{accept}\})$ and $\textsc{safety}(C)$ instead of $\textsc{verify}(C, \{\mathbf{AG}\neg\mathit{error}\})$.

Theorem 23 ([21]). *The two problems* LIVENESS(SN-ASM$_{\text{fct}}$) *and* SAFETY (SN-ASM$_{\text{fct}}$) *are both undecidable. Moreover,* SAFETY(C) *is undecidable for every class C that includes all deterministic ASMs in* SN-ASM$_{\text{fct}}$ *whose external vocabulary is empty and whose dynamic vocabulary consists of two nullary function symbols and arbitrarily many nullary relation symbols.*

The proof of this theorem is by reduction of the halting problem for Turing machines. It is based on the following two observations:

1. Inputs that contain non-nullary functions suffice to encode bit-strings.
2. Two nullary dynamic functions suffice to check whether a bit-string encodes an accepting computation of a Turing machine.

An analysis of the proof indicates that automatic verification of even very simple ASMs whose inputs can be used for encoding bit-strings is not feasible. Nevertheless, Theorem 22 and results in [22] show that restricted variants of ASMs with relational inputs can be verified automatically.

Acknowledgements. Thanks to the anonymous referees.

References

1. A. Blass and Y. Gurevich. Existential fixed-point logic. In E. Börger, editor, *Computation Theory and Logic*, volume 270 of *Lecture Notes in Computer Science*, pages 20–36. Springer-Verlag, 1987.
2. A. Blass and Y. Gurevich. The Linear Time Hierarchy Theorems for Abstract State Machines. *Journal of Universal Computer Science*, 3(4):247–278, 1997.
3. A. Blass, Y. Gurevich, and J. Van den Bussche. Abstract State Machines and Computationally Complete Query Languages. This volume.
4. A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. Technical Report MSR-TR-99-08, Mircosoft Research, 1999.
5. E. Börger and J. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of the EATCS*, 64:105–127, February 1998.
6. D. Beauquier and A. Slissenko. Verification of Timed Algorithms: Gurevich Abstract State Machines versus First Order Timed Logic. Technical Report TIK-Report 87, ETH Zürich, March 2000.
7. E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of 22nd Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM '95)*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer Verlag, 1995.
8. E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic. *ACM Trans. on Prog. Lang. and Sys.*, 8(2):244–263, April 1986.
9. G. Del Castillo and K. Winter. Model Checking Support for the ASM High-Level Language. Technical Report TR-RI-99-209, Universität-GH Paderborn, June 1999.
10. H. D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer-Verlag, 1995.

11. E.A. Emerson. Temporal and Modal Logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–11072. Elsevier Science Publishers B.V., 1990.
12. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. This volume.
13. E. Grädel and M. Spielmann. Logspace Reducibility via Abstract State Machines. In J. Wing, J. Woodcock, and J. Davies, editors, *World Congress on Formal Methods (FM '99)*, volume 1709 of *Lecture Notes in Computer Science*, pages 1738–1757. Springer-Verlag, 1999.
14. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
15. Y. Gurevich. May 1997 Draft of the ASM Guide. Technical Report CSE-TR-336-97, University of Michigan, May 1997.
16. Y. Gurevich. The Sequential ASM Thesis. *Bulletin of the EATCS*, 67:93–124, 1999.
17. N. Immerman and M.Y. Vardi. Model Checking and Transitive Closure Logic. In *Proceedings of 9th International Conference on Computer-Aided Verification (CAV '97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 291–302. Springer-Verlag, 1997.
18. A. Levy, I. Mumick, Y. Sagiv, and O. Shmueli. Equivalence, Query-Reachability, and Satisfiability in Datalog Extensions. In *Proceedings of 12th ACM Symposium on Principles of Database Systems (PODS '93)*, pages 109–122, 1993.
19. E. Rosen. An existential fragment of second order logic. *Archive for Mathematical Logic*, 38:217–234, 1999.
20. A.P. Sistla and E.M. Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the Association for Computing Machinery*, 32(3):733–749, July 1985.
21. M. Spielmann. Automatic Verification of Abstract State Machines. In *Proceedings of 11th International Conference on Computer-Aided Verification (CAV '99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 431–442. Springer-Verlag, 1999.
22. M. Spielmann. Verification of Relational Transducers for Electronic Commerce. In *Proceedings of 19th ACM Symposium on Principles of Database Systems (PODS 2000)*. ACM Press, 2000. To appear.
23. M. Spielmann. *Abstract State Machines: Verification Problems and Complexity*. PhD thesis, RWTH Aachen. In preparation.
24. ASM Web Site. http://www.eecs.umich.edu/gasm. Maintained by J. Huggins.
25. K. Winter. Model Checking for Abstract State Machines. *Journal of Universal Computer Science*, 3(5):689–701, 1997.
26. K. Winter. Methodology for Model Checking ASM: Lessons learned from the FLASH Case Study. This volume.

Towards a Methodology for Model Checking ASM: Lessons Learned from the FLASH Case Study

Kirsten Winter[1]

GMD FIRST, Germany
kirsten@first.gmd.de

Abstract Gurevich's Abstract State Machines (ASM) constitute a high-level specification language for a wide range of applications. The existing tool support for ASM was extended, in a previous work, to provide computer-aided verification, in particular by model checking. In this paper, we discuss the applicability of the model checking approach in general and describe the steps that are necessary to fit different kinds of ASM models for the model checking process. Along the example of the FLASH cache coherence protocol, we show how model checking can support development and debugging of ASM models. We show the necessary refinement for the message passing behaviour in the protocol and give examples for errors found by model checking the resulting model. We conclude with some general remarks on the existing transformation algorithm.

1 Introduction

An ASM model comprises the specification of the state space of the system and its behaviour specified by state transitions. The state space is given by means of universes and functions over these universes. If the domains and ranges of all contributing functions are finite (and not too large) and fix an ASM model can be transformed into a model checker language, e.g. the language of the SMV model checker ([7]). (We call a universe *fix* if it may not be extended during a run by firing some transition rules.)

A first schematic approach is published in [13]. In [2] the schema is extended for coping with ASM with n-ary functions ($n > 0$). All n-ary functions will be unfolded to get 0-ary functions that can be mapped to simple state variables in the SMV model. Of course, model checking as a fully automatic approach is limited with respect to the computational effort and thus not feasible for every ASM model. However, our extended transformation approach can tackle a much broader set of applications and yields a real improvement of the former simple transformation.

Once a model is transformed into the model checker language we benefit from checking properties concerning safety and liveness. If counterexamples can be detected they may yield a good insight in the model under development.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 341–360, 2000.

On the other hand, our transformation tool supplies SMV with a high level modelling language. The modelling task is facilitated by allowing the use of more complex data types and n-ary functions for parametrisation. In general, we found that ASM models, in comparison to similar SMV models (cf. section 6), are more concise but yet more general by means of using parameters. Also, an ASM model can be scaled up more easily than an SMV model, which have to come along with simple state variables rather than n-ary functions.

This process of transforming the ASM model into SMV language, and checking the resulting SMV model, is running automatically, i.e. without user interaction, once safety and liveness properties are formalised in a temporal logic (for SMV this is CTL). This is why the model checking approach is so attractive for industry. No expertise seems to be necessary for using a model checker, it is simply a "press button functionality".

This works not only in principle, it works in practice too, as examples show. In most cases, however, we have to adapt the model at hand in order to fit it to the model checking process.

1.1 Fitting ASM Models for Model Checking Process

Most ASM — on a certain level of abstraction — do not comprise state and state transition specification only, but some additional *assumptions* (cf. fig 1).

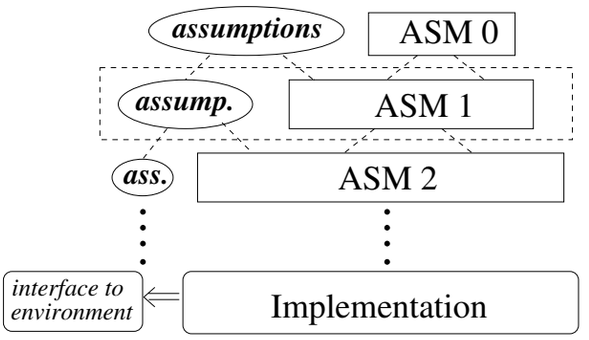


Figure 1. Different Layers of Abstraction

Often these assumptions are necessary for the mathematical proof in order to cope with parts that are abstract or external for the ASM model. They are given informally or formalised by some logic expression in the text that surrounds the specification of the transition rules. Assumptions related to system inherent behaviour will be specified in terms of ASM transition rules at some lower level of abstraction, whereas assumptions over the environment of the system are dedicated to external behaviour that should not be specified further by rules

in a refinement step, but rather stated in a declarative way on a high level of abstraction (e.g. by means of temporal logic formulae). This is necessary for the development of embedded systems that are not regarded to be robust against arbitrary — possibly faulty — environment behaviour. We give some simple examples:

- In the ASM model of the bakery algorithm ([1]) the ticket function T is external and its behaviour is abstract in the most abstract model. But we find that some logical properties are assumed that have to be satisfied by T. This logical properties are then specified in the appropriately refined ASM.
- Also, for the specification of the bakery algorithm, we find that a fairness assumption is necessary for proving correctness of the algorithm ([1]). At the moment there is no feature in ASM language to express fairness assumptions. It should be discussed to extend the language in this direction.
- For embedded systems like the production cell, the specification of the environment is abstract ([8]) and not part of the ordinary transition system. The behaviour of sensors is formalised by means of oracle functions. However, it is necessary to assume that the behaviour of the environment is "reasonable" in order to guarantee correctness of the ASM model. In [8] most assumptions are given in terms of logical formulae that remain external for the refined ASM, too.
- For the specification of protocols we might abstract from the underlying communication model governing the transfer of messages, like in the ASM model of the FLASH cache coherence protocol in [3]. But for the proofs we have to assume that the messages are transfered according to a particular strategy (e.g. the FIFO-strategy, such that the ordering of messages is preserved). At some lower level in the ASM hierarchy a proper message passing behaviour has to be specified that implements the assumption made on the order of transfered messages (cf. section 3).

Obviously, only states, state transitions, *and* assumptions on the model together give the complete specification of the problem at hand. This view is sketched in fig. 1: the dashed box comprises all parts of a model at a particular level of abstraction.

In ASM, external or system inherent but (on a certain level) abstract behaviour is specified by oracle functions. Oracle functions – in case of finite domains – can be simply transformed into corresponding non restricted state variables within the SMV model (neither their initialisation nor their updating is specified), i.e. we give a *loose specification* and a behaviour may be chosen non-deterministically. Since model checking is complete testing over the whole state space, every loose specification leads to a complete case distinction over all possibilities. The problems arising with this are not only a matter of the known state explosion problem, but if we fail to meet the additional assumptions on oracle functions (those which are not expressed by means of transition rules), the SMV model comprises behaviour that is excluded in the *complete* ordinary ASM model.

For our approach of transformation, we have to add the assumptions, given outside of the transition rules, to the SMV model:

- Fairness constraints can simply be added in the SMV code, since the SMV-language offers the corresponding feature.
- For embedded systems, since the current ASM tools do not support the specification of assumptions in terms of some temporal logic, we have to specify assumptions on the environment by means of ASM transition rules to get a suitable input for the transformation. It is a difficult task, however, to model behaviour accurately in a constructive way, because abstract properties have to be encoded operationally. This process easily leads to under-specification or over-specification, i.e. the specification is too open or too restrictive. It must be carefully inspected if the specification meets the assumptions. Otherwise errors may remain undetected or a "wrong counterexample" may be generated (see below). We have to be aware that the results of model checking will hold only with respect to the specified environment.
- Assumptions made on abstract parts of the internal model can simply be added by means of refining the model appropriately. As an example the reader is referred to the case study presented in the following sections. (In particular, the matters of refinement are discussed in section 3).

Let $\mathcal{M}_{ASM}$ the set of ASM models, $\mathcal{M}_{SMV}$ the set of SMV models, and $\mathcal{RUN}$ the set of runs of an ordinary temporal structure over branching time, which provides a semantics that both kinds of model have in common. To give a more precise notion of missing assumptions, we introduce mod and $\overline{mod}$ be two functions that yields the semantics of a model such that $mod : \mathcal{M}_{ASM} \rightarrow \mathcal{RUN}$ and $\overline{mod} : \mathcal{M}_{SMV} \rightarrow \mathcal{RUN}$. Problems may arise in two cases:

$$mod(\mathcal{M}_{ASM}) \subset \overline{mod}(\mathcal{M}_{SMV})$$

$$mod(\mathcal{M}_{ASM}) \supset \overline{mod}(\mathcal{M}_{SMV})$$

In the first case — the transformed SMV-model is strict greater than the ASM-model because of a loose specification of assumptions — $\overline{mod}(\mathcal{M}_{SMV}) \setminus mod(\mathcal{M}_{ASM})$ may contain runs, that violate the property to be checked. One of these may give a counterexample that is, however, not a proper run of the ASM model. No proposition on the model under investigation is possible then since only one counterexample will be given. We call it *wrong counterexample*, it obstructs the overall debugging process.

In the second case — when specifying assumptions too restrictive and the SMV-model is strict smaller than the ordinary ASM-model — we may fail to detect errors that occurs only in those runs of the ASM-model that are excluded for the SMV-model. The runs in $mod(\mathcal{M}_{ASM}) \setminus \overline{mod}(\mathcal{M}_{SMV})$ will not be checked by the model checker.

In the following we give an example for fitting an ASM-model to our model checking approach. We show how the ASM model of the FLASH cache coherence protocol that is given in [3] can be refined in order to add the assumption that

are stated in the proof part. Also, the necessary fairness assumption are added in the SMV code. Assumptions on environmental behaviour are not needed in this particular case, we can keep the "environment" by means of oracles functions without any restricting specification.

In section 2 we present the original abstract model of the FLASH protocol from [3]. Section 3 introduces our refinements that are necessary to complete the model to be transformed. The results that show the benefits from model checking are presented in section 4. In section 5 the transformation and its possible optimisations is discussed. We give remarks on related work in section 6 and conclude with section 7.

2 Case Study: The FLASH Cache Coherence Protocol

The Stanford FLASH multiprocessor (cf. [6]) integrates support for cache coherent shared memory for a large number of interconnected processing nodes. That is, each node (or processor) holds some part of memory, which is accessible for other nodes too, in order to read or write data from this part. One of the problems to be investigated for this architecture is the coherence of the data, since access to data is realized by working with a copy of it. It may happen that one node is going to process a copy of the data when the data have changed in the meanwhile (because of some writing access of another node).

The parts of the distributed memory are given as sets of lines, i.e. small pieces of memory content. Each line is associated with a *home* node hosting the part of the physical memory where the line resides. Whenever a process (or a node) needs to have access to a particular line, we say that a *read* or *write miss* occurs. Every read or write miss concerning a remote memory line triggers a line *request* to its *home* node.

Being interconnected the nodes are able to communicate with each other. They can send (and deliver) messages in order to request for a remote line or to respond on a request by means of sending a copy of the needed data. To provide coherence of the data additional book-keeping is necessary to prevent from simultaneous reading and writing on the same line. That is, writing needs *exclusive* access to a line whereas, reading is allowed in *shared* access. Message passing and book-keeping of shared and exclusive access is the matter of the protocol specification we consider in our ASM model.

The ASM model of the protocol in [3] is based on agents. Each agent models one (processor) node that holds a certain block of the distributed memory. A set of transition rules describes the behaviour of a single agent. Its behaviour is determined by the incoming message that is to be processed, in fact the type of a message is suitable for determination. This notion yields the clear model structure shown in figures 2 and 3.

Incoming messages are requests from remote nodes (or from the receiving node itself in order to simplify the model, this special case is called *intra-node communication* further on). In figure 2 (lower part) the reader may find transition rules to be triggered when a request for shared access is received. In figure 3

(upper part) transition rules for handling requests for exclusive access are depicted. For readability the message types to be distinguished are surrounded by boxes.

A read or write miss that causes a request to be sent is arbitrarily generated by means of oracles. These oracles come along as messages as well, their types have the form **cc.**__ (cf. figure 2, upper part). Whenever the agent receives such kind of message it generates the corresponding request and enters a waiting mode. Releasing an access is handled in the same way. If the agent sends a request for releasing a line the state of the line is invalidated, that is, the node has no reliable copy of the line any more.

In our adaptation of the model the parts related to data (to be sent within the messages) are discarded. Since the data do neither influence the control flow of the protocol behaviour (data do not control any of the guards in the transition rules) nor determine the properties to be checked, we do not lose expressiveness of the model and our checking results.

State Functions. Beside the message type the agent's behaviour depends on several state variables: *curPhase(line)* (phase of the current request), *State(line)* (state of the local line copy in use), and *pending(line)* (flag for currently processed request). *Owner(line)* and the set of *Sharers* of a line are also taken into account. These functions are local for each agent, the additional parameter *self* is omitted in figures 2 and 3.

Message Structure. A message is modelled as a quintuple consisting of the type of the message, the addressed agent, the sender agent, the agent initiating the request and the requested line.

The message types related to **shared** access are:

`get:`	requesting a line from its *home*
`put:`	granting a line to the requester (*source* of the request)
`fwdget:`	forwarding the request to an exclusive owner of the line
`swb:`	requesting a write-back of an owned line that is to be shared
`nack, nackc:`	negatively acknowledging the request or forwarded request (`nackc`), if it cannot be performed now.

The transition rules that are related to requests for shared access are given in figure 2: the circulation of a "get"-request. A get-request may either be (a) negatively acknowledged if another request on the line is already processing (*pending(line)* is true), or (b) it is forwarded to the current owner if there is already an exclusive access (*owner(line)* not undefined), or (c) it is granted to the requester if no owner is noted (else case). If there is an owner (case (b) above), the grant for shared access is given by the agent which "believes" to be an owner, and, moreover, the owner has to release its exclusive copy. If an agent gets a forward get-request and does not agree to be the owner of the line ($State(l)$ is not exclusive) then the request has to be negatively acknowledged as well.

In analogy, message types related to **exclusive** access are:

`getx:`	requesting a line for exclusive access from its *home*
`putx:`	granting a line for exclusive access to the requester

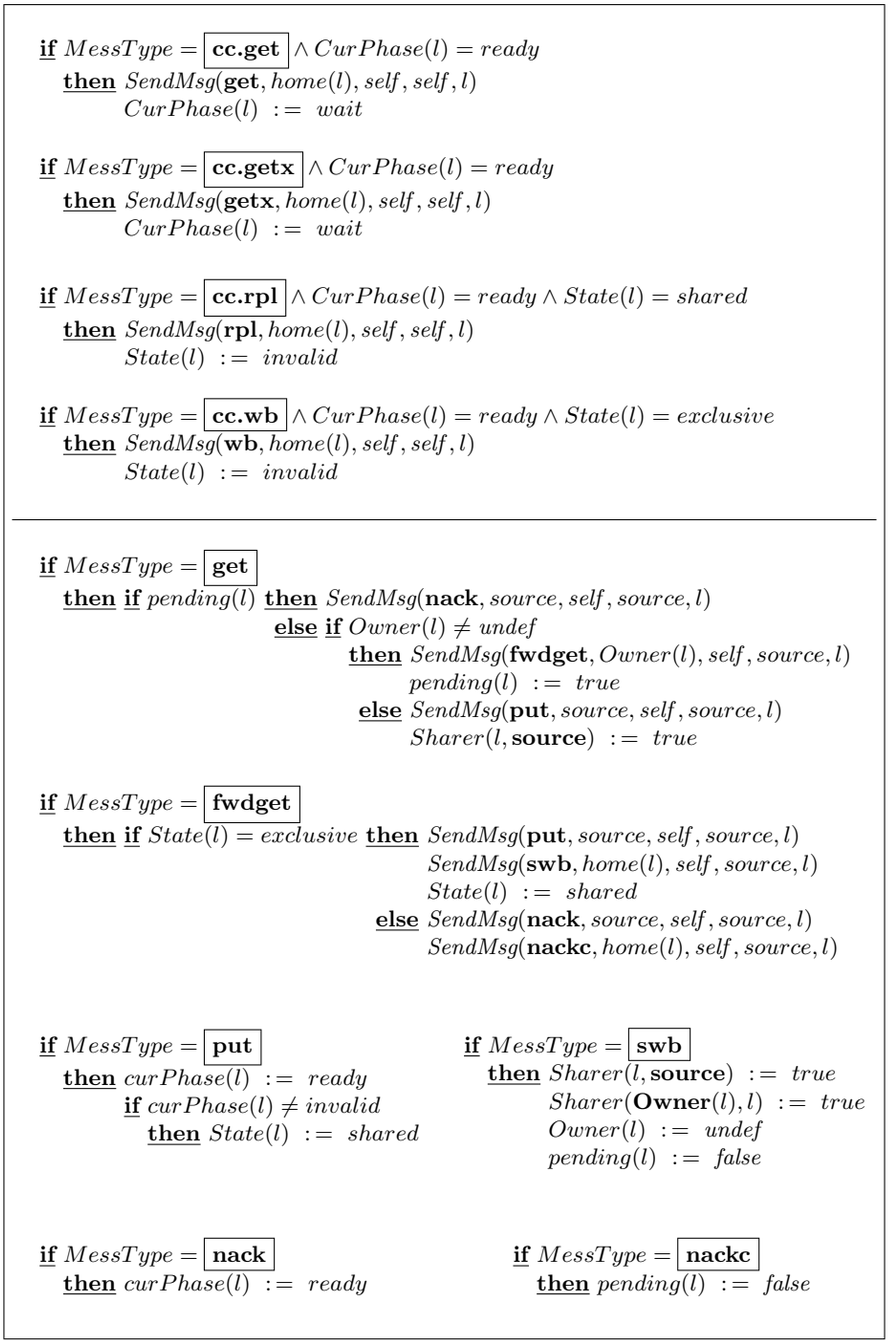


Figure 2. Responding on a request for shared access

```
if MessType = [getx]
  then if pending(l)
         then SendMsg(nack, source, self, source, l)
          else if Owner(l) ≠ undef
                 then SendMsg(fwdgetx, Owner(l), self, source, l)
                      pending(l) := true
                  else if ∃u : Sharer(l, u)
                         then ∀u : Sharer(l, u)SendMsg(inv, u, self, source, l)
                              pending(l) := true
                          else SendMsg(putx, source, self, source, l)
                              Owner(l) := source

if MessType = [fwdgetx]
  then if State(l) = exclusive
         then SendMsg(putx, source, self, source, l)
              SendMsg(fwdAck, home(l), self, source, l)
              State(l) := invalid
          else SendMsg(nack, source, self, source, l)
              SendMsg(nackc, home(l), self, source, l)

if MessType = [putx]
  then State(l) := exclusive
       CurPhase(l) := ready

if MessType = [fwdAck]
  then if Owner(l) ≠ undef
         then Owner(l) := source
              pending(l) := true

if MessType = [inv]
  then SendMsg(invAck, home(l), self, source, l)
       if State(l) = shared
         then State(l) := invalid
          else if curPhase(l) = wait
                 then curPhase(l) := invalidPhase

if MessType = [invAck]
  then Sharer(l, MessSender) := false
       if ∀a : Agents | a ≠ MessSender ∧ Sharer(l, a) = false
         then SendMsg(putx, source, self, source, l)
              pending(l) := false

---

if MessType = [rpl]
  then if ∃u : Sharer(l, u) ∧ ¬pending(l)
         then Sharer(l, u) := undef

if MessType = [wb]
  then if Owner(l) ≠ undef
         then Owner(l) := undef
```

Figure 3. Responding on a request for exclusive access

`fwdgetx:` forwarding the request for exclusive access the owner of the line
`inv:` requesting a current *sharer* of the line to invalidate its local copy
`invAck:` acknowledging the invalidation of the line
`fwdAck:` owner's granting according to a forwarded shared request.

The corresponding behaviour is shown in figure 3: the circulation of a "getx"-request. Again, the exclusive request may be negatively acknowledged if another request is in process already, or it is forwarded to the owner if there is one noted. In addition to the get-request we have to take care for possible sharers if there is no owner. Each of the sharers of the line has to be informed about the request for exclusive access (by sending an `inv`-message). When getting an `inv`-message, sharer has to invalidate its copy of the line and response with an acknowledgement of invalidation (sending `invAck`-message). When each of the sharers has sent its invalidation acknowledgement to home a grant for exclusive access is send to the requester (i.e. sending `putx`-message is delayed until all `invAck`-messages are received).

For releasing a shared or exclusive copy from its cache an agent sends a write_back (`wb`) or a replace message (`rpl`) to *home*. The agent has to be deleted from the list of shares or not being owner any more.

3 Refinements of the ASM Model

Sending a message is given as a macro definition. In the abstract model of [3] *SendMsg* adds a message to a (possibly infinite) set of messages in transit using the `extend` rule constructor. The strategy for receiving a message from this set is not specified. For the proof it is just assumed in [3] that the messages are received in the right order. In order to formalise this assumption and to keep the model finite we had to refine the model.[1] By means of the refinement we restrict the model excluding those runs, in which a message is sent but never received, or sent but overwritten by a message that is sent later. That is, we want to exclude that messages may received in a different order than being sent. Without our refinements the model is either infinite (in [3] the set of messages in transit is not restricted at all) and yield wrong counterexamples as well (if we rely on restricting the set to be finite only). We cannot guarantee liveness if messages may be received in a wrong order.

In order to avoid loss of messages and to guarantee that the order of sending messages and receiving a message is proper, we add a strategy of synchronisation: (a) We introduce a queue for messages in transit for each agent. (b) A message can be send only if the queue for messages in transit is not full. (c) We interleave the protocol behaviour with synchronisation steps. Each step of protocol communication (specified by the rules given in figures 2 and 3) is followed by one step of message passing through. The refinement and necessary changes are described in the following subsections.

[1] At `http://www.first.gmd.de/~kirsten` the listing of the refined ASM-SL model can be found.

3.1 Queue for Messages in Transit

Instead of handling messages as tuples we decided to keep the components of a message as single functions. (That is to avoid unnecessary complexity caused by the unfolding step in the transformation.) The formerly infinite universe `MessInTransit` is replaced by finite queues for each agent that holds one of the message components:

`MessInTr: QLength * Agent -> Type`,
`SenderInTr: QLength * Agent -> Agent`,
`SourceInTr: QLength * Agent -> Agent`, and
`LineInTr: QLength * Agent -> Line`.

The universes `QLength`, `Agent`, and `Line` are sets that are finitely restricted by a maximal index: `maxQ`, `maxAgent`, or `maxLine`. These constants can easily be adjusted in order to scale up the model under investigation.

Sending a message (*SendMsg*) is now specified as appending the message components to the corresponding functions: If n_i is the smallest index for which the queue for messages in transit is empty, we update all message-component-functions at this index. We indicate emptiness of queue(n_i) by means of message type `noMess` for the message type component, i.e. if `(MessInTr(`n_i`,` a_i`)=noMess)` is satisfied and for all $n_j < n_i$ it is not, then n_i marks the index for writing the message to be append. This is specified by means of the following macro:

```
transition AppendToTransit(agent_,(sender_,mess_,source_,line_)) ==
 if MessInTr(1,agent_)=noMess
 then  SenderInTr(1,agent_) := sender_
       MessInTr(1,agent_)  := mess_
       SourceInTr(1,agent_):= source_
       LineInTr(1,agent_)  := line_
 else do forall i in { 2..maxQ }
       if MessInTr((i-1),agent_)!=noMess and MessInTr(i,agent_)=noMess
       then SenderInTr(i,agent_) := sender_
            MessInTr(i,agent_)  := mess_
            SourceInTr(i,agent_):= source_
            LineInTr(i,agent_)  := line_
       endif
      enddo
 endif
```

For requests (i.e. message type is `cc.get`, `cc.getx`, `cc.wb`, `cc.rpl`) we introduce an extra queue for request in transit `MessInTrR`. Analogously, we define dynamic functions that hold one of the components of a request. Sending of a request message is specified by means of the macro `AppendRequestToTransit` that looks similar to the appending macro shown above.

3.2 Passing through of Messages in Transit

Sending of a message by means of appending it to the queue of messages in transit, is only one half of the message exchange. As a second step we have to

specify how the messages in transit are read by the agents to be addressed, that is how messages are passing through (cf. Fig. 4).

Each agent holds a *current message* (which is seen to be empty if its type equals to `noMess`). Again we model the current message by means of its components: `InMess`, `InSender`, `InSource`, and `InLine` are dynamic functions over domain `Agent`. The current message is delivered by passing through the first element of the transit-queues `MessInTr` and `MessInTrR` (see Fig. 4). Note, that requests have lower priority than messages, i.e. a request is passed through only if there is no message in transit left.

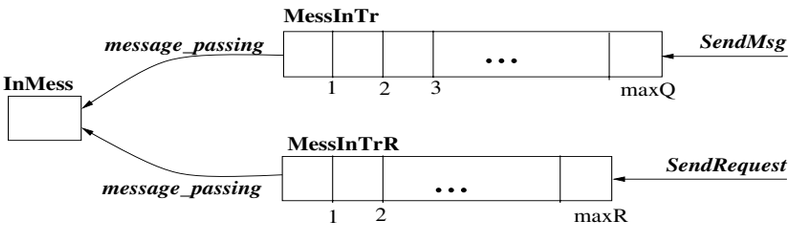


Figure 4. Message passing through from MessInTransit to Incoming Message

Processing the current message and delivering a new message operates on the same dynamic functions of the model. In order to avoid racing we have to interleave both steps: message processing of the agents and message passing through. We extend the overall behaviour by means of a sub-step for synchronisation. In the synchronisation step the messages are passed through to the addressed agent in the proper order.

In an ASM model, introducing a sub-step (to be processed after each state transition) is structure preserving: in addition to the ASM for message processing we specify an ASM for the message passing through. That is, we do not have to interfere the ordinary transition rules (cf. figures 2 and 3) with transition rules for message passing through. An overall ASM `main` invokes both "sub-ASM" `agent_behaviour` and `message_passing` in turn:

```
transition main ==
  if toggle = behave
  then agent_behaviour
       toggle := sync
  else (* if toggle = sync *)
       message_passing
       toggle := behave
  endif
```

Taking this, we benefit from the clear and understandable structure of the abstract model that is given by [3].

3.3 Racing Situation for Intra-node-Communication

Racing situations (i.e. simultaneous updating of the same location through different rules) is excluded since we apply an interleaving semantics to the agents running in parallel, and introduce sub-steps for message passing through (as described above). However, racing may be caused by *intra-node communication*: To keep the ASM model small we do not distinguish whether the requesting node is also home of the line under request or not. That is, if the home of a line wants to have shared or exclusive access of its own line, it has to proceed in the same way as all other nodes has to, i.e. sending a `get`/`getx`-message to itself.

If requester is also home of the line it may happen that two message are to be send simultaneously to the same address. A racing situation occurs, one of the messages (the first of two updates) will be lost. (In figures 2 and 3 the reader may find such situations in the `fwdget`-rule and the `fwdgetx`-rule.)

In order to avoid racing we introduce three new message types that combine two messages: `put_swb`, `putx_fwdAck`, and `nack_nackc`. Whenever the source of the message (i.e. the requester) is equal to the home of the line under request, we send one of combined messages instead of sending two single messages. According to the new message types we introduce also rules that combine both of the rules that have to be fired when receiving such messages.

4 Results of Model Checking

We take model checking of the transformed ASM model as an evolutionary process of debugging: we edit the ASM model, transform it automatically into an SMV model, run SMV to check the properties under investigation, simulate the resulting counterexample (if any) with the ASM Workbench, and edit the ASM model again. As the debugging process is more efficient if the model checking terminates in a reasonable span of time we keep our model in the beginning as small as possible. We find that, even if the model is restricted to two communicating agents and one line of memory, we detect errors in the abstract model as well as in our refinement. In the following we describe *some* of the detected errors as examples.[2] The errors raise when checking the model for **safety** and **liveness**, i.e.:

- No two agents have exclusive access on the same line simultaneously.
- Each request will eventually be acknowledged.
- Whenever an agent gets shared access home will note it as a sharer.

[2] The corresponding listings of the counterexamples can be found in the appendix of the full version of this paper to be find on `http://www.first.gmd.de/~kirsten`.

We formalise these requirements in CTL, e.g.[3]:

$\bigwedge_{i \neq j}$[`AG (!(State(a(i),l)=exclusive & State(a(j),l)=exclusive))`]
$\bigwedge_{i}$[`AG (curPhase(a(i),l) = wait -> AF (curPhase(a(i),l) = ready))`]
$\bigwedge_{i}$[`AG ( State(a(i),l)=shared -> AX (Sharer(l,a(i)) = true)`
`| Sharer(l,a(i)) = true -> AX (State(a(i),l)=shared)) `]

Our first counterexample shows simultaneous exclusive access. We simulate the ASM model and run into an error that can also be found in the abstract ASM model of [3]:

> Whenever a `putx`-message is sent to grant exclusive access the addressed requester has to be noted as owner of the line. This is specified for the `getx`-rule but it is missing for the `invAck`-rule that might also cause a `putx`-message to be sent (see also Fig. 3). The protocol is unsafe since simultaneous exclusive access may occur, and written data may be lost.

Other counterexamples are dedicated to the problem of racing (i.e., conflicts caused by simultaneous updates) on the finite message queue. Although our data space is restricted to a very short queue, we can derive more general remarks for the message passing via queues that concern the finite limitation of a queue itself (which, in fact, is characteristic for each possible realization), e.g.:

> Each sharer of a requested line has to process the rule for invalidation (`inv`-rule). It sends an `invAck`-message to *home* for acknowledging the invalidation. When receiving an `invAck`-message, *home* deletes the sender from the list of sharers. If *home* is sharer too (in case of intra-node communication (see Sec. 3.3)), a deadlock may occur if the number of sharers is greater or equal than the length of the message queue: *home* may fail to complete with the `inv`-rule when the queue is full and sending a message is not possible (since every other sharer may have sent before); *home* stays busy and can not process the incoming `invAck`-rule to clear the queue. In general, we found out that the message queue must be larger or equal than the number of agents since in the worst case each agent is a sharer and will send simultaneously an `invAck`-message to the home node.

We correct our ASM model in that we reply the guards of the request-rules (i.e. rules for `cc.get`, `cc.getx`, `cc.wb`, and `cc.rpl`) when passing the requests in transit through to the home of the requested line.

The examples show that helpful borderline cases can be detected more easily by a model checker than by pure simulation. The computational effort for the automated transformation of our models ranges from three to five seconds. The size of the resulting SMV models is given below where the different columns yield

[3] Though the third specification is rather weak, it yields helpful counterexamples. The disjunction is necessary due to the fact that the sequence in that the corresponding updates occur (noting a state to be shared and adding a sharer) changes for different scenarios.

the comparing results when scaling up the parameters for the number of agents and lines.[4] The variable ordering is determined by the automatic reordering facility that is given by the SMV.

resources used:	2 agents, 1 line	3 agents, 1 line	2 agents, 2 lines
user time/system time:	4.69 s/0.13 s	5687.52 s/0.6 s	17263.2 s/0.86 s
BDD nodes allocated:	70587	1612740	2975127
Bytes allocated:	4849664	37748736	54657024
BDD nodes repr. transition relation:	19261 + 78	288986 + 82	78365 + 96

Although checking our model of the FLASH protocol is only feasible for a small number of agents and lines, the results show that the counterexamples yield extremely helpful scenarios for locating errors.

5 Transformation Issues

The general approach for the transformation of ASM transition rules into SMV code is already described in [13]. The extended transformation algorithm that includes the transformation of n-ary dynamic functions (with $n > 0$) can be found in [2]. Based on this foundation we want to add some more general remarks here concerning extensibility and optimisation. In order to make this paper self containing we recall our main ideas of transformation.

5.1 Transformation of ASM into SMV Language

The basic translation scheme introduced in [13] can be applied to transform into SMV a subset ASM_0 of ASM, where: *(i)* only **nullary** dynamic and external functions are allowed; *(ii)* the only available data types are integers, booleans and enumerated types; *(iii)* the only defined static functions are those corresponding to predefined operations in SMV (boolean operations, +, -, etc.).

As the semantic models for ASM_0 are essentially basic transition systems, the translation of ASM into SMV is very close:

- non-static functions (i.e., dynamic and external functions) are identified with locations and thus mapped one-to-one to SMV state variables;
- values of the ASM data types are mapped one-to-one to SMV constants;
- applications of static functions are translated to applications of the corresponding built-in operators of SMV.

What remains to be done is to restructure the ASM program into a form where updates of the same location, together with their guards, are collected together. This is done in two steps. First, we transform an ASM program P into an equivalent ASM program P' consisting only of a block of guarded updates (i.e., rules of the form `if` G `then` $f(\bar{t}) := t$) by means of a "flattening" transformation:

[4] The experiments were carried out on an UltraSPARC-II station with 296MHz and 2048 Mb memory, the operating system is Solaris 2.6.

$$[\![R_T]\!] = \begin{cases} \texttt{if } G_T^1 \texttt{ then } R_T^1 \\ \dots \\ \texttt{if } G_T^n \texttt{ then } R_T^n \end{cases} \quad [\![R_F]\!] = \begin{cases} \texttt{if } G_F^1 \texttt{ then } R_F^1 \\ \dots \\ \texttt{if } G_F^m \texttt{ then } R_F^m \end{cases} \Rightarrow [\![\texttt{if } G \texttt{ then } R_T \texttt{ else } R_F]\!] = \begin{cases} \texttt{if } G \wedge G_T^1 \texttt{ then } R_T^1 \\ \dots \\ \texttt{if } G \wedge G_T^n \texttt{ then } R_T^n \\ \texttt{if } \neg G \wedge G_F^1 \texttt{ then } R_F^1 \\ \dots \\ \texttt{if } \neg G \wedge G_F^m \texttt{ then } R_F^m \end{cases}$$

Second, we collect all guarded updates of the same location, thus obtaining, for each location *loc* occurring on the left-hand side of an update in P', a pair $(loc, \{(G_1, t_1), \dots, (G_n, t_n)\})$ which maps *loc* to a set of pairs (guard, right-hand side). Such a pair is translated into the following SMV assignment:

```
ASSIGN next(C⟦loc⟧) :=
    case C⟦G1⟧ : C⟦t1⟧ ;  ...  C⟦Gn⟧ : C⟦tn⟧ ;  1 : C⟦loc⟧ esac;
```

where $\mathcal{C}[\![.]\!]$ denotes here the ASM $\rightarrow$ SMV compiling function for terms, which is straightforward for ASM_0.

The ordinary transformation shown above is extended in order to tackle the ASM language as complete as possible. We have to exclude, however, infinite domains as well as the notion of `import` or `export` transition rules because the domains of a model have to be fix and may not grow during a run. But any other rule constructor as well as arbitrary data types and operations (in particular, lists, finite sets, finite maps and user-definable freely generated types, as provided by ASM-SL) can be used without restriction. Finite quantifications are also supported.

In our approach we reduce an arbitrary (finite and fix) ASM to ASM_0. The main problem here is that in general we do not know which location is updated by $f(t_1, \dots, t_n) := t$ (if $n > 0$) because the parameters t_i may be dynamic functions and thus can change their current value. Evaluation of a location is possible if and only if all t_i are evaluated, i.e. either they are static or have to be unfold to cover all possible values. Thus, the basic idea of the transformation is to iteratively unfold and simplify rules until all terms can be reduced to values or locations.

Terms can be simplified by means of the transformation $[\![.]\!]_\rho$ defined in Table 1, which is then extended to rules in a canonical way.

The rule-unfolding transformation $\mathcal{E}$, which operates on closed rules such as the program P, is formally defined in Table 2. It works as follows:

- if the rule R consists of a block of update rules of the form *location* := *value*, it terminates and yields R as result (there is nothing left to unfold);
- otherwise, it looks for the first location l occurring in R (but not as left-hand side of some update rule) and unfolds R according to the possible values of l. In turn, the unfolding has to be applied to the sub-rules $[\![R[\,l/x_i]]\!]$ obtained by substituting the values x_i for l in R and simplifying.

Applying $\mathcal{E}$ to the (simplified) ASM program $[\![P]\!]_\emptyset$ yields a program $P' = \mathcal{E}([\![P]\!]_\emptyset)$ which is essentially an ASM_0 program.

Table 1. Term and Rule Simplification

Term Simplification

$[\![x]\!]_\rho = x \qquad [\![\, l \,]\!]_\rho = l$

$$[\![\, v \,]\!]_\rho = \begin{cases} x = \rho(v) & \text{if } v \in \mathrm{dom}(\rho) \\ v & \text{otherwise} \end{cases}$$

$[\![t_i]\!]_\rho = x_i$ for each $i \in \{1, \ldots, n\} \;\Rightarrow$

$$[\![f(t_1, \ldots, t_n)]\!]_\rho = \begin{cases} x = \mathbf{f}(x_1, \ldots, x_n) & \text{if } f \text{ static function name} \\ l = (f, (x_1, \ldots, x_n)) & \text{if } f \text{ dynamic/external function name} \end{cases}$$

$[\![t_i]\!]_\rho = l$ or $[\![t_i]\!]_\rho = f'(\vec{t}')$ for some $i \in \{1, \ldots, n\} \;\Rightarrow$
$[\![f(t_1, \ldots, t_n)]\!]_\rho = f([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho)$

$$[\![(Q\ v\ \texttt{in}\ A : G)]\!]_\rho = \begin{cases} [\![G]\!]_{\rho[v \mapsto x_1]}\ op\ \ldots\ op\ [\![G]\!]_{\rho[v \mapsto x_n]} \\ \qquad \text{if } [\![A]\!]_\rho = x = \{x_1, \ldots, x_n\} \quad \text{(i.e., if } [\![A]\!]_\rho \text{ is a value)} \\ (Q\ v\ \texttt{in}\ [\![A]\!]_\rho : [\![G]\!]_{(\rho \setminus v)}) \\ \qquad \text{otherwise.} \end{cases}$$

(where $op \equiv \wedge$ if $Q \equiv \texttt{forall}$, $op \equiv \vee$ if $Q \equiv \texttt{exists}$).

Rule Simplification

$[\![\texttt{skip}]\!]_\rho = \texttt{skip}$

$[\![t_L := t_R]\!]_\rho = [\![t_L]\!]_\rho := [\![t_R]\!]_\rho$

$[\![R_1 \ldots R_n]\!]_\rho = [\![R_1]\!]_\rho \ldots [\![R_n]\!]_\rho$

$$[\![\texttt{if}\ G\ \texttt{then}\ R_T\ \texttt{else}\ R_F]\!]_\rho = \begin{cases} [\![R_T]\!]_\rho & \text{if } [\![G]\!]_\rho = \mathbf{true} \\ [\![R_F]\!]_\rho & \text{if } [\![G]\!]_\rho = \mathbf{false} \\ \texttt{if}\ [\![G]\!]_\rho\ \texttt{then}\ [\![R_T]\!]_\rho\ \texttt{else}\ [\![R_F]\!]_\rho & \text{otherwise.} \end{cases}$$

$[\![\texttt{do forall}\ v\ \texttt{in}\ A\ \texttt{with}\ G\ \ R']\!]_\rho =$

$$= \begin{cases} [\![\texttt{if}\ G\ \texttt{then}\ R']\!]_{\rho[v \mapsto x_1]}\ \ \ldots\ \ [\![\texttt{if}\ G\ \texttt{then}\ R']\!]_{\rho[v \mapsto x_n]} \\ \qquad \text{if } [\![A]\!]_\rho = x = \{x_1, \ldots, x_n\} \quad \text{(i.e., if } [\![A]\!]_\rho \text{ is a value)} \\ \texttt{do forall}\ v\ \texttt{in}\ [\![A]\!]_\rho\ \texttt{with}\ [\![G]\!]_{(\rho \setminus v)}\ \ [\![R']\!]_{(\rho \setminus v)} \\ \qquad \text{otherwise.} \end{cases}$$

Table 2. Rule Unfolding

Rule Unfolding

If R has the form $l_1 := x_1 \ldots l_n := x_n$, then $\mathcal{E}(R) = R$.
Otherwise:

$$\begin{aligned} \mathcal{E}(R) = \ & \texttt{if}\ \ l = x_1\ \texttt{then}\ \mathcal{E}([\![R[l/x_1]]\!]_\emptyset) \\ & \texttt{else if}\ \ l = x_2\ \texttt{then}\ \mathcal{E}([\![R[l/x_1]]\!]_\emptyset) \\ & \ldots \\ & \texttt{else if}\ \ l = x_n\ \texttt{then}\ \mathcal{E}([\![R[l/x_n]]\!]_\emptyset) \end{aligned}$$

where l is the first location occurring in R (but not as lhs of an update rule) and $\{x_1, \ldots, x_n\}$ is the range of location l.

5.2 Interfacing Other Model Checkers

Using the SMV as a tool for checking ASM was the choice for our first approach in this field. This decision was guided by the fact that the semantics of the SMV modelling language (i.e. the target language for the transformation) is given by simple transition systems. That is, it is very close to the ASM semantics. In principle, the SMV can easily be substituted by any other model checker that is based on transition systems, too, e.g., the SVE ([4]) or the VIS ([5]). Since our transformation consists of several steps:

1. Unfolding the n-ary dynamic functions in order to get a simple ASM with only 0-ary functions (named ASM_0).
2. Flattening the nested structure of the transition rules of the ASM_0 in order to get sets of simple guarded updates for each location (i.e. sets of triples (*location*, *guard*, *update_value*)).
3. Transforming the individual sets of guarded updates into our notion of abstract syntax of SMV language.
4. Pretty printing the model given in abstract syntax of the SMV language into SMV code.

In order to develop an interface to any other model checker that treats a kind of transition system it is obviously sufficient to substitute only the last two steps of the transformation.

5.3 Optimisation Issues

Since efficiency is a crucial point for the model checking approach we also have to focus on optimisation issues. The results of the investigations so far are summarised in the following two points:

Location Unfolding. In [2] the rule unfolding is formally given by the schema shown in section 5.1 (see table 2).

For the implementation of this schema we identified a crucial point for optimisation: The set of locations to be unfolded (i.e. substituted by their values) should be more restricted. We regard all boolean operators, e.g. not, and, or, equal as *primitive functions*. Only those locations that occur as function parameters of non-primitive functions or appear at the right hand side of updates or equations are to be substituted. Locations that appear as 0-ary functions on the left hand side of equations (i.e. in guards) or as parameters for primitive functions should not be unfolded. Although this optimisation seems to be obvious it is crucial for our approach in order to keep the transformation efficient (and feasible) even for large models. Since the transformation schema given in [2] suggests the exhaustive use of induction (even more since we implement our transformation by means of ML) it appeared that additional case distinctions are crucial. With the resulting optimised transformation algorithm we are able to generate SMV code in few seconds. Thus, the transformation is not any longer the bottleneck of our approach.

Module Structure in SMV Code. In the current approach for the transformation we choose the most simple way for treating agents in ASM: We consider *self* (denoting the data space of a particular agent) as a simple oracle

function that is updated non-deterministically. The resulting SMV code has a plain structure since we do not exploit the notion of modules or processes given in the SMV language. Instead, the transition behaviour of all agents is specified to be parallel and synchronous but guarded by the current value of *self*. Using a more elaborated module structure in the target language we might benefit from the notion of interleaving semantics of parallel process. In [7] it is stated that the internal BDD structure (that represents the model) may be smaller in size for interleaving asynchronous than for synchronous transition relations.

To investigate a possible optimisation of the transformation, we edited the SMV code of our running example in that we introduced the model structure appropriately for specifying interleaving agents. We compared the resulting SMV models in their size of BDD, which is one of the measurements for the model checking algorithm. (The other one is the CTL formula to be checked.) The corresponding order of SMV state variables is produced by `-reorder` option of SMV.

- without use of processes, normal transformation:
  ```
  BDD nodes allocated: 55446
  Bytes allocated: 6029312
  BDD nodes representing transition relation: 12444 + 54
  ```
- with use of process but only global variables:
  ```
  BDD nodes allocated: 76367
  Bytes allocated: 3473408
  BDD nodes representing transition relation: 11649 + 50
  ```
- with use of process and with locally declared process variables:
  ```
  BDD nodes allocated: 89365
  Bytes allocated: 3670016
  BDD nodes representing transition relation: 13626 + 50
  ```

At least for our example of the FLASH protocol we found that the size of BDDs that represent the corresponding models are comparable. Therefore we keep the transformation without exploiting the notion of SMV modules in order to keep the implementation as simple as possible.

6 Related Work

In [11] a different approach for automated verification for ASM is elaborated. Spielmann represents an ASM model independently of its possible input by means of a logic for computation graphs (called CGL*). The resulting formula is combined with a CTL*-like formula which specifies properties and both are checked by means of deciding their finite validity. This addresses the problem of checking systems with arbitrary inputs. However, Spielmann concludes that this approach is restricted to ASM with only 0-ary dynamic functions and relational input. For many problems, however, we found that the input, or the environment respectively, proper to the ASM model at hand is restricted by certain assumptions (as described in Sec. 1.1). The restriction to 0-ary functions outweighs this

potential benefit. In [10] some helpful theoretical results of the model checking and the verification problem for ASM are published. This investigation tends to define the limit of both problems in general in terms of their decidability and complexity.

In some cases the formalisation of an environment behaviour that is restricted by assumptions is a complex task. Operational semantics (like ASM) may yield more complex solutions than a logical approach (like the temporal logic LTL). It might be useful to facilitate the user with LTL for the specification of additional assumptions on the external behaviour. This approach is supported by the model checker SVE ([4]). Thus, adapting our transformation to the SVE tool might be promising.

The case study of FLASH protocol that is used here in order to give an example for applying our approach is investigated within other approaches as well. The most elaborated work is published in [9]. Park and Dill introduced a method relying on so called *aggregation function* that yields the refinement relation between specification and implementation. Their approach for proving correctness is supported by means of the interactive theorem prover PVS. Giving up on fully automated support they can tackle a broader range of requirements, of course.

Wing and Vaziri-Farahani investigated in [12] a slightly similar kind protocol for cache coherence (but not exactly the FLASH protocol) by means of model checking, namely using SMV. As we do they claim that "*Without loss of generality, we can analyse this cache coherence protocol by considering one client, C, one server, S, and one file, f*'. But the introduced SMV model, in contrast to the ASM model we used, lacks the possibility to scale up easily by using parameters. We found the ASM model, being more general but yet concise at the same time, leads to a more clear and understandable structure than modelling in plain SMV language only.

Also McMillan in [7] applied the SMV to a distributed cache protocol. But his SMV model of the protocol being spread over several modules is not fully introduced and thus difficult to understand. Since the application is not exactly the same as the FLASH protocol we cannot relate the effort for checking (BDD size and checking time) properly.

7 Conclusions

In previous work ([13] and [2]) an interface from the ASM Workbench to SMV was presented. The transformation from ASM to the SMV language comprises the treatment of dynamic functions of arity $n > 0$, which is an crucial improvement, as most ASM specifications benefit from the abundant use of parametric dynamic functions.

In this paper we gave an overview of our experience with model checking ASM so far. Although, our transformation can treat a great subset of the ASM language we found that in practice most ASM models have to be refined or extended to fit for the model checker: Since assumptions on the ASM model that

are stated besides the ordinary transition rules are not covered by the transformation we have to add a proper specification of these assumptions into the resulting SMV-model. This can be done by simply exploiting language features of SMV (e.g. the fairness construct), by further refinement of the ASM model at hand, or by specifying environment behaviour by means of ASM transition rules such that the given assumptions are covered by the transformation.

The practicability of the approach is demonstrated by a non-trivial case study: the ASM model of the FLASH protocol. In order to fit the model for the model checking process we have to refine the model in that we specify the message passing behaviour. We gave some examples for errors found by the model checker that can hardly be detected by pure mathematical proofs, and deduce more general constraints for the model at hand from the counterexamples.

We conclude with some remarks on the implementation of the transformation algorithm concerning extensibility and optimisation issues.

Acknowledgements. My thank belongs to the anonymous referees for their helpful comments as well as Stephan Herrmann and Giuseppe Del Castillo for many fruitful discussions.

References

1. E. Börger, Y. Gurevich, and D. Rosenzweig. The Bakery Algorithm: Yet Another Specification and Verification. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.
2. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. on 6th Int. Conf. TACAS 2000*, volume 1785 of *LNCS*, pages 331–346, 2000.
3. A. Durand. Modeling cache coherence protocol - a case study with FLASH. In U. Glässer and P. Schmitt, editors, *Procs. of the 28th Conf. of German Society of Computer Science*, TR, Magdeburg University, 1998.
4. T. Filkorn et al. *SVE Users' Guide*. Siemens AG, München, 1996.
5. The VIS Group. Vis: A system for verification and synthesis. In T. Henzinger R. Alur, editor, *8th Int. Conf. CAV'96*, volume 1102 of *LNCS*, 1996.
6. J. Kuskin, D. Ofelt, and M. Heinrich et. al. The stanford FLASH multiprocessor. In *21th Int. Symp. on Computer Architecture*, 1994.
7. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
8. L. Mearelli. An Evolving Algebra Model Of The Production Cell. Master's thesis, Universita di Pisa, 1996.
9. S. Park and D. Dill. Verification of cache coherence protocols by aggregatiuon of distributed transactions. *Theory of Computing Systems*, 31:355–376, 1998.
10. M. Spielmann. Model Checking Abstract State Machines and Beyond. In *This Volume*.
11. M. Spielmann. Automatic verification of abstract state machines. In N. Halbwachs and D. Peled, editors, *Computer Aided Verification, CAV '99*, number 1633 in LNCS, pages 431–442, Trento, Italy, 1999.
12. J. M. Wing and M. Vaziri-Farahani. A case study in model checking software systems. *Science of Computer Programming*, 28:273–299, 1997.
13. K. Winter. Model checking for abstract state machines. *J.UCS Journal for Universal Computer Science (special issue)*, 3(5):689–702, 1997.

Report on a Practical Application of ASMs in Software Design

Egon Börger[1], Peter Päppinghaus[2], and Joachim Schmid[2]

[1] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it
[2] Siemens AG, Corporate Technology, D-81730 Munich, Germany
{peter.paeppinghaus, joachim.schmid}@mchp.siemens.de

Abstract. ASMs have been used at Siemens Corporate Technology to design a component in a software package called FALKO. Main purpose of FALKO is the construction and validation of timetables for railway systems. For simulation the whole closed-loop traffic control system is modelled within FALKO. The railway process model part of FALKO was formally specified using the ASM approach. C++ code is generated from the formal specification and compiled together with the handwritten C++ code of the other components to obtain the FALKO executable. The project started in May 1998 and was finished in March 1999. Since then FALKO is used by the Vienna Subway Operator for the validation of the whole subway operational service.

1 FALKO

Abstract State Machines (ASMs) [3,1] have been used at Siemens Corporate Technology to design a component in a software package called FALKO. This name is the German acronym for "Timetable Validation and Timetable Construction" describing concisely the main functionality of this tool. Detailed operational timetables including e.g. vehicle roster plan can be automatically calculated from raw data like train frequency and infrastructure of a railway system. For these calculations estimations of trip times are used. Timetables – whether constructed with FALKO or other tools – have to be validated for operability and robustness. Conventionally this is done by (physically) driving trial runs. With FALKO this costly form of validation can be replaced by dynamic simulation of operational timetables modelling quantitative aspects like velocities and trip times as accurately as needed.

To perform dynamic simulation of timetables, the whole closed-loop traffic control system is modelled within FALKO. The model is designed in a modular way with three main components: train supervision/train tracking, interlocking system, and railway process model. Software components for train supervision/train tracking and for the interlocking system are nowadays part of real train operation. The railway process model on the other hand serves to replace the real physical system (trains, signals, switches etc.) in the simulation.

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 361–366, 2000.

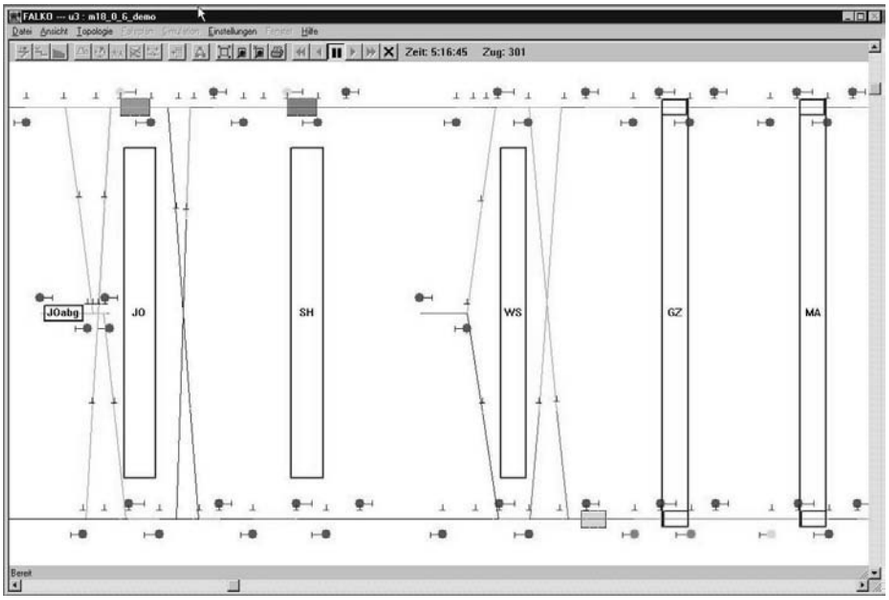


Fig. 1. Screenshot of FALKO GUI

The modelling is based on discrete event simulation. The three components communicate via events, which are tagged with certain data and stamped with the (fictitious) time of their occurrence. Let us give an example. Train supervision, e.g., sends an event at time t_0 to the interlocking system requesting route r_3. The interlocking system, possibly after having received an event which frees route r_3, sends an event at time t_1 to the railway process model requesting switch sw_5 to change its position. The railway process model then calculates the duration of rotating this switch and sends an event to the interlocking system to the effect that switch sw_3 has arrived at the requested position at time t_2, etc.

Besides the three main components modelling the traffic control system there is as a fourth hidden component the event handler, which controls the simulation by receiving events from the visible components and sending them to the respective addressee in the appropriate order.

The railway process model is based on a physical model of driving according to which the crucial numerical calculations for determining speeds, trip times etc. are done. This physical model is another hidden component of FALKO realized as a C++ library.

Further parts of FALKO are a comfortable GUI and extensive facilities to analyze and graphically display data of a simulation run. The GUI can also be used for visualization of simulation runs (see figure 1).

Fig. 2. Screenshot of HTML documentation

2 Design of Railway Process Model with ASMs

2.1 Design Process and Development Tools

The railway process model has been designed with ASMs [3,1]. All the other components of FALKO have been designed and implemented conventionally with handwritten C++ code.[1] Along with this design a prototypical development environment has been built up, which can also be used to maintain this component of FALKO.

This development environment supports a seamless flow from the specification down to the executable code. The single source for FALKO's railway process model is a specification consisting of formal parts together with informal explanations. The formal parts are ASM rules and static, dynamic, derived and external functions written in ASM-SL, the language of the ASM workbench [2]. The specification comes as a collection of HTML documents (see figure 2). Hyperlinks to navigate between uses and definitions of rules and functions are generated automatically. The formal parts can be extracted automatically from the HTML documents to be fed into the front end of the ASM workbench, which is used for syntax and type analysis.

[1] The physical model of driving is not included in the railway process model. It has been designed conventionally as a component of its own.

In the design phase of the project the ASM workbench was also used for early tests of the ASM model(s). At the end of the design phase it was decided to attempt code generation rather than hand coding the already debugged ASM model.

In the implementation phase a code generator has been developed, automatically generating C++ code from type correct ASM-SL code. In addition some wrapper code for interfacing the generated code to the remaining components, and some low-level "library code" was hand coded.

Formal verification of the ASM model has not been attempted, this was not a goal of the project.

2.2 Effort for Design and Implementation

Design Phase

- Requirement specification based on predecessor system, developed in meetings of the design team, documented by minutes of the meetings (4 persons 2 weeks)
- Design of 1^{st} draft of executable ASM model (1 person 8 weeks)
- Several cycles of testing and debugging using the ASM workbench [2] (1 person 8 weeks + 1 person 11 weeks)
- Review of 2^{nd} draft of ASM model by design team plus external reviewers (6 persons 1 week)
- Several cycles of improving, testing and debugging (2 persons 5 weeks)

Implementation Phase

- Development of ASM-SL to C++ code generator (1 person 4 weeks)
- Specification and implementation of additional handwritten C++ code (1 person 2 weeks)
- Integration of FALKO system including testing and debugging (3 persons 3 weeks)
- Documentation of railway process model component and final polish (1 person 6 weeks)

Summing up that part of the above listed effort, which was spent on behalf of the railway process model component of FALKO, this yields a total effort of 66 person weeks.

It is not possible, of course, to compare this effort reliably to the corresponding effort in a conventional software design process. But a rough estimation done by an experienced programmer intimately familiar with FALKO says that the ASM design (including development of the C++ code generator) has exceeded a conventional effort by about 10%.

2.3 Size of ASM Model and C++ Code

ASM Model (source of C++ code generation)

- ca. 3 000 lines of ASM workbench code
- 120 rules
- 315 functions and relations (240 functions, 75 relations)
  - 71 dynamic
  - 69 external
  - 59 static
  - 116 derived

C++ Code

- ca. 9 000 lines of generated C++ code
- ca. 2 900 additional lines of handwritten C++ code, consisting of
  - ca. 400 lines wrapper code for interfacing to other components of FALKO
  - ca. 2 500 lines low-level library code

In the prototypical predecessor system of FALKO the railway process model consisted of ca. 20 000 lines of (handwritten) C++ code. To be fair one has to take into account, however, that this component – having been experimented with – had grown over time (and become difficult to maintain, which was the reason for redesigning it completely).

3 Experiences

It turned out that one of the main advantages of ASMs for this design was the parallel update view. This made it possible to model the railway process in such a way that each state of the ASM model corresponds to a "snapshot" of the virtual physical process taken at the occurrence of a "relevant discrete event". Due to this, one can always have a clear picture of the physical state snapshot, on which auxiliary computations (specified by static and derived functions) are based.

FALKO developers and reviewers not concerned with formal methods had no problems to understand the ASM model. The possibility of early tests by executing the ASM model with the ASM workbench was very helpful and uncovered bugs also in other components of FALKO at an early stage.

No serious attempt has been made to measure the potential performance loss due to the use of generated C++ code. Comparison to the predecessor system of FALKO has been possible only on one example, for which the data happened to be available in both systems. In this small example performance of the previous system was about 30% better than that of the current FALKO system. For the time being the performance problem has been left aside, since it turned out that FALKO's performance is good enough for the purpose the product is used for.

FALKO is used in four installations at the Vienna Subway Operator since March 1999, one of these installations being in daily use. Up to now (i.e. March

2000) the customer reported no bugs. After finishing the first version of FALKO, two bugs in the railway process model have, however, been discovered in tests during development of the second version. The FALKO developers have not yet familiarized themselves with the ASM specific tools. The generated C++ code being readable enough they chose to implement temporary fixes of the two bugs by handhacking the generated C++ code, and to postpone correction of the ASM model until the faults are analyzed more thoroughly.

Acknowledgments. We thank our colleagues of the FALKO team for their open attitude towards this pioneering use of ASMs and for a year of enjoyable and fruitful cooperation. Furthermore we thank Giuseppe Del Castillo for the permission to use early versions of his ASM workbench and for the support we obtained from him.

References

1. E. Börger and J. Huggins. Abstract State Machines 1988-1998: Commented ASM Bibliography. *Bulletin of EATCS*, 64:105–127, February 1998. Updated bibliography available at http://www.eecs.umich.edu/gasm.
2. Giuseppe Del Castillo. *The ASM Workbench – A tool environment for computer aided analysis and validation of ASM models.* PhD thesis, University of Paderborn, to appear.
3. Yuri Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

Using Abstract State Machines at Microsoft: A Case Study

Mike Barnett, Egon Börger*, Yuri Gurevich,
Wolfram Schulte, and Margus Veanes

Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
{mbarnett, boerger, gurevich, schulte, margus}@microsoft.com

Abstract. Our goal is to provide a rigorous method, clear notation and convenient tool support for high-level system design and analysis. For this purpose we use abstract state machines (ASMs). Here we describe a particular case study: modeling a debugger of a stack based runtime environment. The study provides evidence for ASMs being a suitable tool for building *executable* models of software systems on various abstraction levels, with precise refinement relationships connecting the models. High level ASM models of proposed or existing programs can be used throughout the software development cycle. In particular, ASMs can be used to model inter component behavior on any desired level of detail. This allows one to specify application programming interfaces more precisely than it is done currently.

1 Introduction

This paper describes a case study on the use of ASMs as support for design and analysis of software at Microsoft. In order to use ASMs we needed a tool for executing ASMs integrated with the Microsoft programming environment, in particular with the *Component Object Model* or *COM* [5]. We developed a prototype called AsmHugs [8] by extending the Hugs system [2] which is an implementation of the lazy functional programming language Haskell. AsmHugs is in many ways similar to AsmGofer [1] but Hugs enabled us to use H/Direct [6, 7] for integration with COM. A detailed technical report of this case study is in preparation [3].

1.1 What Is the Case Study about?

We present a model for a command-line debugger of a stack-based runtime environment. The model was reverse-engineered from the debugger which is written in C++ and which uses a particular application programming interface (API). We have in fact three models of the debugger, each at a different level of abstraction. They form a refinement hierarchy where each refinement relationship is a procedural abstraction.

* Visiting researcher from Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy, boerger@di.unipi.it

Y. Gurevich et al. (Eds.): ASM 2000, LNCS 1912, pp. 367–379, 2000.

1.2 Why This Particular Case Study?

Our motivation was to study the applicability (and integrability) of ASMs to Microsoft software. Software at Microsoft is generally written in modules, or components. These often form client/server relationships where a component may function both as a server and as a client. A client and a server interact solely through an established API. An API is a set of procedures to which all communication is restricted. The debugger in question is an application program for end-users. It is a client of the *debug services*.

Suppose you want to understand how the API works without reading the code. You have a specification in the Interface Definition Language (IDL [5]) which gives you only the relevant *signatures*. The inner workings of the API's methods are hidden; the only additional information you may find is an informal natural-language description. Such descriptions may be incomplete and become often inconsistent with the code, as the code evolves over time. Obviously, there is no way to formally enforce correspondence between code and its natural-language description. The two main problems with using program code itself as documentation are these. First, the code is usually huge and includes too many irrelevant details. Second, the code might not be available for proprietary reasons.

In this study, we model a particular program but our interests are broader: how to use ASMs to better specify the APIs by which different components interact. An ASM model provides the missing information at the appropriate level of abstraction so that the user of a given component can understand both the behavior of the component and the protocol that the user is supposed to follow in order to exploit the behavior. Each method is specified as a rule, and the valid patterns of calls are reflected in the state.

As part of a broader project, we also built an ASM model of the other side of the API: the debug services (to be described in a forthcoming report); the debugger model was a valuable source in that context.

1.3 COM

Microsoft software is usually composed of COM components. These are really just static containers of methods. In your PC, you will find dynamic-link libraries (DLLs); a library contains one or more components (in compiled form). COM is a language-independent as well as machine-independent binary standard for component communication. An API for a COM component is composed of *interfaces*; an interface is an access point through which one accesses a set of methods. A client of a COM component never accesses directly the component's inner state, or even cares about its identity; it only makes use of the functionality provided by different methods behind the interface (or by requesting a different interface).

1.4 Integration with COM

Writing an executable model for a client of a COM component requires that either you model also the server side of the interfaces (and then maybe the COM components for which that server is a client, and so on) or you integrate your model into the COM environment and thus make it possible for your model to actually call the COM component. We do the latter. H/Direct provides a means for a Haskell program to communicate with COM components both as a client and as a server. For the debugger model, both modes (the client and the server) are needed. The server mode is needed because the debug services API specifies the *callback* interface that the client must implement. The debug services use the callback interface to make asynchronous procedure calls to the model. However, Hugs is a sequential system. In order to use it in an asynchronous multi-threaded COM environment, a modification of the Hugs runtime was required. Other modifications of the Hugs runtime were needed to allow the outside world to invoke Hugs functions.

All three of the debugger models in the case study, even the most refined one (the *ground* model), are sequential ASMs (enriched with some Haskell functionalities). The ground model communicates with the outside world. The question arises how to view that communication in ASM terms. Calls from the ground model to the debug services are invocations of external functions. Calls in the other direction (the callbacks) happen to be more specific: callbacks are updates of nullary monitored functions.

2 The Case Study

The case study is to model a sample debugger of a particular runtime environment. The main goal of this debugger is to illustrate a correct way to use the debugging services provided by that runtime. See Figure 1. Nonetheless, the debugger has more than 30k lines of C++ code and exhibits complex behavior, partly due to the involvement of several asynchronous agents (the user, the runtime and the operating system), but mostly because of the complexity of the debug services that expose about 50 interfaces and 250 methods.

A careful analysis of the debugger, mainly by analyzing the source code, led us to a refinement hierarchy consisting of three models. By analyzing the causal dependencies of different actions we realized that the complications due to the seemingly asynchronous behaviour could be completely avoided, which enabled us to model the debugger by a *sequential* ASM. The design decisions which are reflected by the resulting debugger model were later validated by running the model as a replacement of the actual debugger.

1. *Control model.* The abstraction level of this model is at the control level. The user can enter a command if the debugger is in a mode where the user has a prompt. The runtime environment can issue a callback to the debugger only if the latter is expecting a callback. At this level, the only effect of a callback or user command is to change the control state.

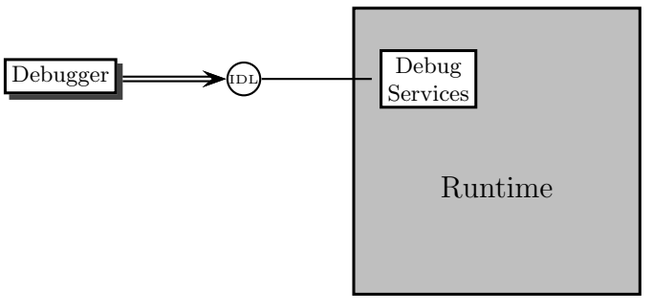


Fig. 1. Debugger and the runtime debug services.

2. *Object model*. This model reflects the *compile time* or *static* structure of the underlying architecture, namely that modules contain classes, classes contain functions, functions contain code, etc. Furthermore, it provides a restricted view of the *run time* structure, namely that processes contain threads, threads contain frames, etc. At this level exactly those commands are modeled that are intimately connected to the compile time structure and to the restricted view of the run time structure, such as execution control, breakpoint control, and stepping commands.
3. *Ground Model*. This model has the same core functionality as the debugger. It provides a more detailed view of the run time structure than the object model. User commands dealing with inspection of the run time stack, and contents of individual frames, like inspection of specific object fields, are modeled here.

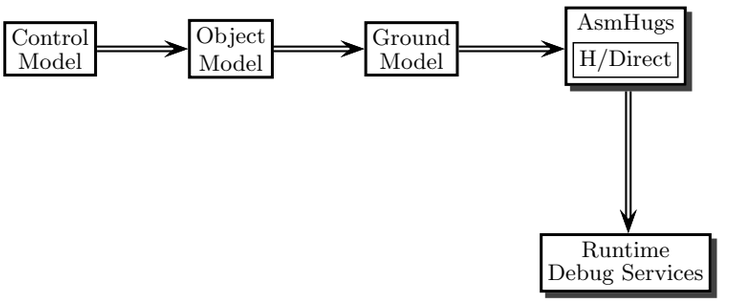


Fig. 2. Connections between the models and the runtime.

In principal, all models are executable, either without the runtime or through the refinements using the actual runtime (see Figure 2). However, in the former case, a proper simulation harness has to be provided. An example of a test harness by using a "wizard" is illustrated in Section 3.2. In a related project, we constructed an ASM model that approximates the runtime. This model could be used here to embody the runtime in the harness.

When executing the control model using the runtime, each action is interpreted by AsmHugs via the indicated refinements; COM connectivity is provided by H/Direct.

3 Control Model of the Debugger

In this model we consider the main control modes of the debugger. These modes reflect who has the control: the user, the runtime, or the debugger itself. When the user has control, the debugger presents a prompt and waits for input. The debugger waits for a response from the runtime when the runtime has the control. Once the runtime has responded with an event, the debugger has control and has to decide what mode to transit to. As we will see, this decision is based on particular properties of the event and further input from the runtime.

Although the communication between the real debugger and the runtime is asynchronous, the documentation of the debug API specifies that the runtime issues at most one asynchronous event at a time; before issuing the next event the runtime must receive an acknowledgement from the debugger. Furthermore, the communication protocol between the debugger and the runtime ensures that at most one of them may have control. Therefore, *sequential* ASMs suffice for our modeling purposes.

The remainder of the section is structured as follows. First, we will lay out the control model ASM in full detail, introducing the necessary state components and the rules as needed. Second, we will run a particular user scenario that will indicate a possible discrepancy between the runtime and the model. Finally, we will remedy the model.

This example shows a typical use of a high level model during the design phase in a software development cycle.

3.1 Control Model ASM

The debugger can be in one of four modes. In `Init` mode the debugger either hasn't been started yet, or it has been terminated. In `Break` mode the user has the control. In `Run` mode the runtime has the control. In `Break?` mode the debugger has the control. The dynamic ASM function, `dbgMode`, records the mode of the debugger in the current state; it has the initial value `Init`. The top level rule of the debugger is `dbg`. Below, `do` is short for `do in-parallel`.

```
dbgMode = initVal Init
dbg = do
```

```
  if dbgMode == Break or dbgMode == Init then handleCommands
  if dbgMode == Run                      then handleResponses
  if dbgMode == Break?                   then handlePendingEvents
```

There are two monitored functions, `command` and `response`, that are updated by the user and the runtime, respectively. Initially, both monitored functions have a value that indicates that no input has been entered by either.

```
command = initVal "nothing"
response = initVal "nothing"
```

User commands. We can partition user commands into three groups: commands for starting and quitting the debugger, commands that hand the control over to the runtime (e.g. execution control and stepping commands), and commands that do not affect the control mode (e.g. state inspection and breakpoint setting).

The user can issue commands only if the debugger is either in `Init` mode or in `Break` mode. In the first case, the only meaningful action is to start the debugger.

```
handleCommands = do onStart
                    onExit
                    onBreakingCommand
                    onRunningCommand
                    command := "nothing"

onStart = if dbgMode == Init  and command == "start"
            then do doCommand("start")
                    dbgMode := Break
```

In `Break` mode the debugger handles normal debugging commands and it may switch to `Run` mode or back to `Init` mode, depending on the command.

```
onExit  = if dbgMode == Break and command == "exit"
             then do doCommand("exit")
                     dbgMode:= Init

onBreakingCommand = if dbgMode == Break and isBreakingCommand(command)
                      then do doCommand(command)

onRunningCommand  = if dbgMode == Break and isRunningCommand(command)
                      then do doCommand(command)
                              dbgMode:= Run
```

Firing of execution control commands and stepping commands implies that the control is handed over to the runtime.

```
isRunningCommand(x) = x in? {"run <pgm>", "continue", "kill",
                             "step into", "step over", "step out"}
```

Other commands have no effect on the control mode.

```
isBreakingCommand(x) =
   not(isRunningCommand(x))  and  x != "exit" and x != "start"
```

As mentioned above, in the control model, handling of commands is a skip operation. This rule is refined in the object model.

```
doCommand(x) = skip
```

Callbacks. The debugger can handle callbacks or responses from the runtime only in the `Run` mode. The value of the monitored function `response` is a callback message from the runtime notifying the debugger about a runtime event. In the debugger, each event is classified either as a *stopping* event or as a *non-stopping* event.

```
handleResponses      = do onStoppingEvent
                          onNonStoppingEvent
                          response := "nothing"

onStoppingEvent    =
  if isStoppingEvent(response) then do dbgMode:= Break?
                                       doCallback(response)

onNonStoppingEvent =
  if not(isStoppingEvent(response)) then doCallback(response)
```

Breakpoint hit events, step complete events, and process exit events are always stopping events. Initially `isStoppingEvent` is the following unary relation (unary Boolean function).

```
isStoppingEvent(x) = x == "step completed" or x == "breakpoint hit" or
                     x == "process exited"
```

However, the relation is dynamic and may evolve during a run as a result of a specific user command or a specific runtime event.

In the control model the actual handling of callbacks is a skip operation. This rule is refined in the object model.

```
doCallback(x) = skip
```

Pending Events. In `Break?` mode a stopping event has happened and the debugger should hand the control over to the user. This happens only if there are no *pending* events in the runtime. Otherwise the control goes back to the runtime and the debugger continues the current process

```
handlePendingEvent = do onPendingEvent
                        onNoPendingEvent

onPendingEvent     =  if isEventPending then do dbgMode := Run
                                                isEventPending := False
                                                doCommand("continue")

onNoPendingEvent   =  if not(isEventPending) then do dbgMode := Break
```

The boolean function `isEventPending` is monitored by the runtime. The control model is summarized by a state diagram in Figure 3.

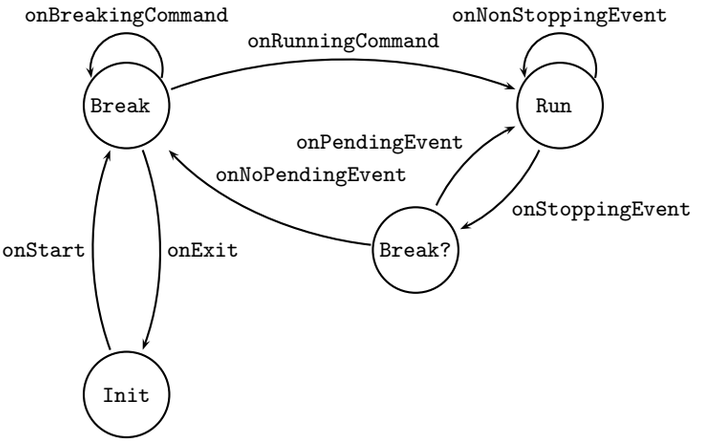


Fig. 3. Control model of the debugger.

3.2 A Wizard-of-Oz Experiment

We want to explore the behavior of the model on a given set of user scenarios, without having access to the real runtime, and possibly expose contradictions in the model. Since we reverse-engineered the debugger, we cannot claim that our model is truly faithful to it. In fact, any error that might show up may very well have sneaked into the model without being in the debugger. The point we want to make is that this form of testing *would* be useful if used during the design phase.

Since the actual runtime is missing, we will ask a "wizard" to play its role. Table 1 shows a run, with row i indicating the part of the state that has changed after the i'th ASM step of the model.

Table 1. A run of the control model.

	dbgMode	command	response	isEventPending
0:	Init	start		
1:	Break	bp hello.cpp:7		
2:	Break	run hello.exe		
3:	Run		created process	
4:	Run		loaded module	
5:	Run		created thread	
6:	Run		hit breakpoint	
7:	Break?			True
8:	Run		loaded class	
9:	Run		...	

The run shows that after we hit a breakpoint there is a pending event in the runtime. According to the model, the current process is continued and the control is passed to the runtime. It turns out that the pending event was a non-stopping event (class was loaded). Obviously, this behaviour contradicts the expected consequence of reaching a breakpoint, namely that the user should get the control. At this point we have two options to solve the problem: if we can constrain the runtime to meet the restriction that only stopping events can cause the `isEventPending` flag to become true, then the current model is correct; otherwise we have to modify the model. We choose the latter, see Figure 4.

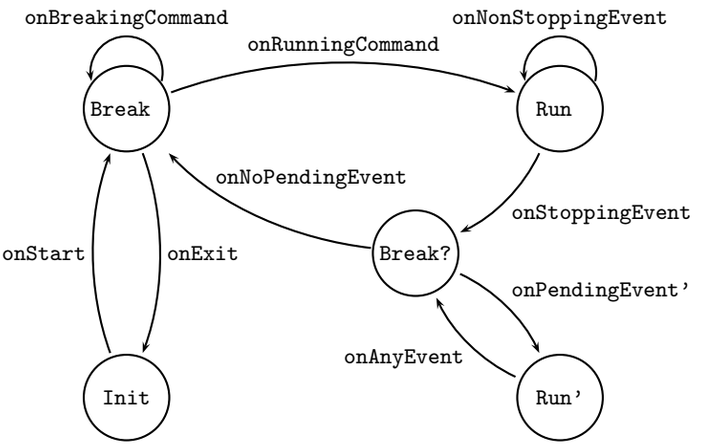


Fig. 4. New control model of the debugger.

4 Object Model

The static and the dynamic structures of the runtime architecture are reflected in the object model with just enough detail so that one can model features that deal with execution control, breakpoints, and stepping, and explore their interaction.

Let us consider some examples. The first example is the refinement of the user command that starts the debugger. The next two examples specify what happens in the object model when a module is loaded or unloaded in the runtime. Recall that all those rules are just `skip` rules in the control model.

4.1 User Commands

Starting the debugger

```
doCommand("start") = do in-sequence
  coInitialize
  shell := newShell(services    = newServicesInterface(...),
                    callback    = newCallbackInterface(...),
                    modules     = {},
                    breakpoints = {},
                    debugee     = undef)
  shell.services.setCallback(shell.callback)
```

The first rule initializes the COM environment. The second rule creates a new debugger shell with a pointer to the services and a new callback. The third rule invokes the setCallback method of the debug services with the new callback interface pointer as argument, thus providing the services access to the client's callback methods.

4.2 Callbacks

Loading of modules

```
doCallback(<"load module",mod>) = do
  shell.modules := shell.modules ++ {mod}
  do in-sequence
     forall bp in shell.breakpoints do bp.bind(mod)
     shell.debugee.continue()
```

The new module is recorded in the shell. The shell attempts to bind each of its set of breakpoints to that module. Once all new bindings have been made (if any), the debugee is continued through an external call.

We have omitted the `bind` rule that checks if the location that the breakpoint refers to indeed exists in the module. If the location exists, a real breakpoint is created at that location through an external call to the services; otherwise, nothing is done.

Unloading of modules

```
doCallback (<"unload module",mod>) = do
  shell.modules := shell.modules \ {mod}
  do in-sequence
    forall bp in shell.breakpoints do bp.unbind(mod)
    shell.debugee.continue()
```

The effect of unloading a module is to update the shell and to remove all the breakpoints from that module.

4.3 Some Comments

There are a couple of things worth noticing about the above rules. First, the design decision to bind a breakpoint that may already be bound implies that if there is a breakpoint referring to a location that exists in two or more distinct modules, then the breakpoint is associated to all of them. Second, all breakpoints are handled simultaneously; there are no ordering constraints between them. This is a typical situation: in the actual (C++) code there is a linearly ordered structure maintaining the elements that are then processed sequentially in the order determined by the structure.

When constructing the object model we detected a mismatch between the way loading and unloading of module callbacks was implemented. Namely, although loading followed the specification above, during unloading of a given module, if a breakpoint was bound to this module, then that breakpoint was not only removed from this module but from *all other* modules as well. We did not discover this when studying the code, but only when constructing the object model. In fact, it is hard to see it from the code, but readily apparent from the model.

5 Ground Model

The ground model has the same core functionality as the debugger and can be executed as an AsmHugs program that communicates with the runtime by using the H/Direct generated glue code. It gives a more detailed view of the run time structure than the object model. All the user commands, such as inspection of specific object fields, are modeled here. The ground model is fully described in the technical report [3].

6 Conclusion

The case study illustrates some particular ways that ASMs can help the system designer:

- *Concisely describe complex systems.* The debugger involves several asynchronous agents and involves about 50 interfaces and about 250 methods. The size of the debugger is about 30K lines of C++ code. The size of the ASM specification (which can be run, though not as fast, and which provides essentially the same functionality) is only 4K lines. Due to built-in parallelism, the runs of our ASM are shallow. In fact it takes only bounded many steps (less than 10) to process one user command in contrast to the C++ code which may require unbounded many steps for the purpose (the number of steps may depend on the program being debugged). The parallelization was obtained, e.g., by abstraction from irrelevant sequentialization in the C++ code (see Section 4.3).
- *Abstract away the environment as needed.* We could easily separate the models, but still formally tie them via refinements, without losing executability.
- *Interactively explore the design on all abstraction levels.* It is very difficult to detect at the source code level such high level bugs as the ones that we detected with relative ease with the Wizard-of-Oz experiment. The object model enabled us to detect inconsistencies in the callback management.

To repeat, our goal is to provide a rigorous method, clear notation and convenient tool support for high-level system design and analysis. Our main tool will execute ASMs (and will help you to write ASMs). Several issues that arose during the case study have influenced the design of that tool.

- We found the parallel synchronous construct `forall` very useful. Similar conclusions were drawn from the Falko project [4].
- Set and list comprehensions turned out to be very convenient.
- We found object oriented notation useful to structure specs, to improve their readability, and to link their execution to the object-oriented programming paradigm.
- We realized that in order for ASMs to be useful in Microsoft (or indeed anywhere COM is used), ASM models must achieve full COM interoperability.

The first and the third point are illustrated by the examples in Section 4.2.

Acknowledgments. We thank Sigbjorn Finne for helping us with H/Direct during the course of this work.

References

1. AsmGofer. `http://www.tydo.de/AsmGofer/`.
2. Hugs98. `http://www.haskell.org/hugs/`.
3. Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using ASMs at Microsoft: A case study. Technical report, Microsoft Research, Redmond, USA, 2000.
4. Egon Börger, Peter Päppinghaus, and Joachim Schmid. Report on a practical application of ASMs in software design. In *This Volume*.

5. Don Box. *Essential COM.* Addison-Wesley, Reading, MA, 1998.
6. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. H/Direct: A binary foreign language interface for Haskell. In *Proc ACM Sigplan International Conference on Functional Programming (ICFP'98), Baltimore*, pages 153–162, 1998.
7. Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In *Proc ACM Sigplan International Conference on Functional Programming (ICFP'99), Paris, France*, 1999.
8. Foundations of Software Engineering, Microsoft Research. AsmHugs. `http://www.research.microsoft.com/foundations/`.

Author Index