Lecture Notes in Computer Science 3052

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA

Josef Kittler
University of Surrey, Guildford, UK

Jon M. Kleinberg
Cornell University, Ithaca, NY, USA

Friedemann Mattern
ETH Zurich, Switzerland

John C. Mitchell
Stanford University, CA, USA

Oscar Nierstrasz
University of Bern, Switzerland

C. Pandu Rangan
Indian Institute of Technology, Madras, India

Bernhard Steffen
University of Dortmund, Germany

Madhu Sudan
Massachusetts Institute of Technology, MA, USA

Demetris Terzopoulos
New York University, NY, USA

Doug Tygar
University of California, Berkeley, CA, USA

Moshe Y. Vardi
Rice University, Houston, TX, USA

Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany
Abstract state machines (ASM) sharpen the Church-Turing thesis by the consideration of bounded resources for computing devices. They view computations as an evolution of a state. It has been shown that all known models of computation can be expressed through specific abstract state machines. These models can be given in a representation-independent way. That is one advantage of transferring these models to ASM. The main advantage is, however, to provide a unifying theory to all of these models. At the same time ASM can be refined to other ASMs. Stepwise refinement supports separation of concern during software development and will support component-based construction of systems thus providing a foundation of new computational paradigms such as industrial programming, programming-in-the-large, and programming-in-the-world.

ASM 2004 continued the success story of the ASM workshops. Previous workshops were held in the following European cities: Taormina, Italy (2003); Dagstuhl, Germany (2002); Las Palmas de Gran Canaria, Spain (2001); Monte Verita, Switherland (2000); Toulouse, France (1999); Magdeburg, Germany (1998); Cannes, France (1998, 1997); Paderborn, Germany (1996); and Hamburg, Germany (1994). The ASM workshops have had predecessors, e.g., the famous Lipari Summer School in 1993, whose influential outcome was the fundamental Lipari Guide.

The success story of the ASM workshops is based upon a number of advantages of the ASM method for high-level design, analysis, validation, and verification of computing systems:

– The specification method improves industrial practice by proper orchestration of all phases of software development, by supporting high-level modeling at any level of abstraction, and by providing a scientific and formal foundation for systems engineering. All other specification frameworks known so far only provide a loose coupling of notions, techniques, and notations used at various levels of abstraction.

By using the ASM method, a system engineer can derive a general application-oriented understanding, may base the specification on a uniform algorithmic view, and may refine the model until the implementation level is achieved. The three ingredients needed to achieve this generality are the notion of the ASM itself, the ground-model techniques, and the proper treatment of refinement.

Using this specification method the system architect, the application engineer, the developer, and the programmer obtain a common view of the system they are building, changing, maintaining, or documenting. The system construction process is accompanied by common understanding, by common theoretical foundations, and by the ability to prove validity of properties such as satisfaction of quality criteria. By doing so the construction process supports quality from the very beginning of the process until the implementation of the system.
At the same time, the ASM method supports software development in changing environments. The method supports abstraction and extension of models, stepwise detailing of models, and control through execution of the model for their experimental validation.

- Abstract state machines entirely capture all four principles of computer science: structure, evolution, collaboration, and abstraction.

This coverage of all principles has not been achieved in any other approach of any other discipline of computer science. Due to this coverage, the ASM method underpins computer science as a whole.

- the ASM method is clearly based on a number of postulates restricting evolution of systems. For instance, sequential computation is based on the postulate of sequential time, the postulate of abstract state, and the postulate of bounded exploration of the state space. These postulates may be extended to postulates for parallel and concurrent computation, e.g., by extending the last postulate to the postulate of finite exploration.

The generality of the model, the deep foundation, the test of the concept by providing rigorous semantics to a large variety of real-life software and hardware products, and – at the same time – the development of proper tools supporting the entire specification process is the work of a community headed by two congenial friends – Yuri Gurevich and Egon Börger – competing at the same time with their results. The former decided to prove the concept in real industrial praxis by building up a group at Microsoft Research, whilst simultaneously deepening the theory of ASM. Beyond further development of the ASM theory, the latter attracted a number of researchers for a program that will entirely change computer science. It will improve development and implementation of languages by providing rigorous semantics and by acquiring tools to prove and to maintain properties of the developed products, and will thus support quality at each level of software specification. The community is growing. The success story can be traced at the website http://www.eecs.umich.edu/gasm/.

ASM 2004 contributed to ASM research in several ways:

Extending ASM foundations: ASM research has brought up a good number of difficult and open problems in computer science. This volume contributes by
providing solutions to slicing ASM, intra-step interaction, theory of monodic ASMs, and interchange languages for ASM. Abstract state machines have recently extended to turbo abstract state machines. This abstraction mechanism has already been used in theory of computation. Furthermore, transition theory contributes to foundations of ASM.

**Highlighting new application areas:** Cryptographic machines, security logics, and service specification are currently hot research areas. As demonstrated in this volume, ASM methods may substantially improve understanding in these areas. Furthermore, the volume shows that even .NET models can be based on ASM.

**Tackling problems in already proven application areas:** Programming language semantics is based on ASM methods for C# and SSA. UML needs more foundations and ASM methods can be used to formalize UML diagrams, e.g. sequence diagrams. Timed systems are one of the areas where ASM methods have successfully been used. The workshop continues to provide a deeper insight into embedded systems and their semantics. ASM semantics has already been used for exploring database behavior. This workshop provides a deeper view on query processing.

ASM 2004 invited four distinguished researchers to bring other aspects to ASM research. Yuri Gurevich extended the postulates of sequential computation to interaction. Hans-Michael Hanisch showed how ASM research might contribute to research on embedded control systems. Hans Langmaack discussed associations between ALGOL semantics and Turbo ASM introduced recently. Jan Van den Bussche demonstrated how database processing may be based on finite cursor machines.

The LNCS proceedings are accompanied with local proceedings demonstrating recent research in ASM development. These papers demonstrate the development of tools, give an insight into ongoing projects, and provide results on ASM theory that may lead to kernel papers of the next ASM workshop.

We thank the members of the program committee and the additional reviewers for their support in evaluating the papers submitted to ASM 2004. We thank both Springer-Verlag for publishing the proceedings with the invited and research papers in the LNCS series and the Martin-Luther-University Halle-Wittenberg for publishing the proceedings of the short papers. We appreciate the diligent service of the organization team: Thomas Kobienia, Michael Schaarschmidt, and Ramona Vahrenhold. We thank our colleagues and the students of our universities for their help in workshop organization. We thank Martin-Luther-University Halle-Wittenberg Microsoft Research, the Stiftung Leucorea, and the Winzervereinigung Freyburg-Unstrut eG for their support of the workshop. Last, but not least, we thank the participants of ASM 2004 for having made our work useful.

March 2004
Wolf Zimmermann
Bernhard Thalheim
Program Co-chairs

Wolf Zimmermann (Martin-Luther-University Halle-Wittenberg, Germany)
Bernhard Thalheim (University of Kiel, Germany)

Program Committee

Egon Börger (University of Pisa, Italy)
Alessandra Cavarra (Oxford University, UK)
John Derrick (University of Kent, UK)
Uwe Glässer (Simon Fraser University, Canada)
Elvinia Riccobene (University of Catania, Italy)
Robert Stärk (ETH Zurich, Switzerland)
Peter H. Schmitt (Universität Karlsruhe, Germany)
Anatol Slissenko (University of Paris, France)
Bernhard Thalheim (Kiel University, Germany)
Margus Veanes (Microsoft Research, Redmond, USA)
Wolf Zimmermann (Universität Halle-Wittenberg, Germany)

Additional Reviewers

E. Asarin D. Beauquier P. Caspi
P. Cegielski A. Chou A. Cisternino
J. Cohen R. Farahbod N. G. Fruja
A. Gargantini R. Ge S. Graf
T. Gross Y. Gurevich J. Jacky
M. Kardos L. Nachmanson S. Nanchen
A. Prinz H. Rust P. Scandurra
M. Schaarschmidt W. Schulte M. Vajihollahi

Organization Committee

Wolf Zimmermann University Halle-Wittenberg
Ramona Vahrenhold University Halle-Wittenberg
Michael Schaarschmidt University Halle-Wittenberg
Thomas Kobienia Cottbus University of Technology

Sponsoring Institutions of the ASM 2004 Conference

Martin-Luther-University Halle-Wittenberg, Germany
Microsoft Research, Redmond, WA, USA
Stiftung Leucorea, Germany
Winzervereinigung Freyburg-Unstrut eG, Germany
Volume Editors

Wolf Zimmermann
Martin-Luther-University Halle-Wittenberg, Germany
Department of Mathematics and Computer Science
Institute of Computer Science
von-Seckendorff-Platz 1, D-06099 Halle/Saale
E-mail: zimmer@informatik.uni-halle.de

Bernhard Thalheim
Kiel University
Institute of Computer Science and Applied Mathematics
Olshausenstr. 40, D-24098 Kiel, Germany
E-mail: thalheim@is.informatik.uni-kiel.de
## Table of Contents

### Invited Papers

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intra-step Interaction</td>
<td>1</td>
</tr>
<tr>
<td>Yuri Gurevich</td>
<td></td>
</tr>
<tr>
<td>Closed-Loop Modeling and Related Problems of Embedded Control Systems in Engineering</td>
<td>6</td>
</tr>
<tr>
<td>Hans-Michael Hanisch</td>
<td></td>
</tr>
<tr>
<td>An ALGOL-View on Turbo ASM</td>
<td>20</td>
</tr>
<tr>
<td>Hans Langmaack</td>
<td></td>
</tr>
<tr>
<td>An ASM Specification of C# Threads and the .NET Memory Model</td>
<td>38</td>
</tr>
<tr>
<td>Robert F. Stärk and Egon Börger</td>
<td></td>
</tr>
<tr>
<td>Finite Cursor Machines in Database Query Processing</td>
<td>61</td>
</tr>
<tr>
<td>Jan Van den Bussche</td>
<td></td>
</tr>
</tbody>
</table>

### Research Papers

<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formalizing Liveness-Enriched Sequence Diagrams Using ASMs</td>
<td>62</td>
</tr>
<tr>
<td>Alessandra Cavarra and Juliana Küster-Filipe</td>
<td></td>
</tr>
<tr>
<td>Specification and Validation of the Business Process Execution Language for Web Services</td>
<td>78</td>
</tr>
<tr>
<td>Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi</td>
<td></td>
</tr>
<tr>
<td>Monodic ASMs and Temporal Verification</td>
<td>95</td>
</tr>
<tr>
<td>Michael Fisher and Alexei Lisitsa</td>
<td></td>
</tr>
<tr>
<td>Towards an Interchange Language for ASMs</td>
<td>111</td>
</tr>
<tr>
<td>Elvinia Riccobene and Patrizia Scandurra</td>
<td></td>
</tr>
<tr>
<td>Specification and Implementation Problems for C#</td>
<td>127</td>
</tr>
<tr>
<td>Nicu G. Fruja</td>
<td></td>
</tr>
<tr>
<td>An ASM Semantics for SSA Intermediate Representations</td>
<td>144</td>
</tr>
<tr>
<td>Sabine Glesner</td>
<td></td>
</tr>
<tr>
<td>Observations on the Decidability of Transitions</td>
<td>161</td>
</tr>
<tr>
<td>Yuri Gurevich and Rostislav Yavorskiy</td>
<td></td>
</tr>
<tr>
<td>A Security Logic for Abstract State Machines</td>
<td>169</td>
</tr>
<tr>
<td>Stanislas Nanchen and Robert F. Stärk</td>
<td></td>
</tr>
</tbody>
</table>
Slicing Abstract State Machines ........................................ 186
    Antje Nowack

The Cryptographic Abstract Machine ................................ 202
    Dean Rosenzweig and Davor Runje

Modeling Discretely Timed Systems Using Different Magnitudes
of Non-standard Reals ................................................... 218
    Heinrich Rust

Author Index ................................................................. 235
Intra-step Interaction

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

Abstract. For a while it seemed possible to pretend that all interaction between an algorithm and its environment occurs inter-step, but not anymore. Andreas Blass, Benjamin Rossman and the speaker are extending the Small-Step Characterization Theorem (that asserts the validity of the sequential version of the ASM thesis) and the Wide-Step Characterization Theorem (that asserts the validity of the parallel version of the ASM thesis) to intra-step interacting algorithms.

1 Why Intra-step Interaction

According to the Lipari guide [9], an abstract state machine interacts with the environment by means of external functions. “The computation steps of a program are supposed to be atomic at an appropriate level of abstraction. A computation step is hardly atomic if during that step the [ASM] queries an oracle and then, depending on the result, submits another query to the same or a different oracle. Thus it seems reasonable to forbid nesting of external functions. Indeed, the need to nest external functions has not arisen in applications so far. But we withhold final judgment and wait for more experimentation.”

It was a good idea to withhold final judgment. Many things happened in the meantime. Probably the most consequential — as far as intra-step interaction is concerned — is the development of AsmL, the Abstract State Machine Language [2]. It is not unusual for an AsmL program π to nest external functions. For example, π may have a do-in-parallel form where one of the constituents calls another agent a′, gets a callback from a′, calls a′ again, etc. and thus executes a nontrivial protocol, possibly involving additional agents. All this is done in a single step. (And if you break the protocol into several steps then the do-in-parallel structure is all messed up.)

One may pretend that every interaction between the algorithm and the environment is inter-step. That point of view is taken in [10]. But, as the amount of intra-step interaction increases, it gets harder and less natural to maintain the pretense. Of course, an omniscient environment can know ahead of time what external function calls the algorithm will generate during a step. It can thus prepare the answers and store them, so that the algorithm would treat external functions as internal. But a typical environment is not omniscient. And if the environment is omniscient, does it need our algorithm at all?

The time came to recognize the importance of intra-step communication and to deal with it directly. Notice that we have been dealing with intra-step commu-
nication all along, albeit implicitly. Consider for example the import command as in the Lipari guide.

```plaintext
import v
R(v)
endimport
```

What really happens here is that the program sends an import query to the environment, and the environment returns a reserve element. The creation of objects in object-oriented languages like C# [8] or Java [11] is similar (when viewed on the abstraction level of the program). Here is another example involving one of the AsmL choose constructs:

```plaintext
any x | x in {1, 2, 3} where x > 1
```

To evaluate this expression, the program sends out a choose query and provides a parameter \{2, 3\}. The environment replies with an element of \{2, 3\}.

In the rest of this extended abstract, an algorithm is intra-step interactive unless the contrary is stated explicitly.

## 2 Terminology

We restrict attention to sequential time algorithms. Here sequential time means that the computation proceeds in a sequence of discrete steps [10]. We recall some useful terminology introduced recently in [4] and used already in [5].

**Small-Step Algorithms.** We quote from [5]: “Small-Step Algorithms are characterized by two properties:

- (Step) The computation proceeds in a sequence of discrete steps.
- (Small) The amount of work done by the algorithm in any one step is bounded; the bound depends only on the algorithm, not on the state, nor on the input, nor on the actions of the environment.

Many authors call such algorithms sequential (for example the present authors in [3,10]), but other authors use ‘sequential’ to mean only the first of these properties (which we call _sequential time_). Because of this ambiguity, we now prefer the term ‘small-step’.”

**Wide-Step Algorithms.** We quote from [4]: “The term ‘parallel algorithm’ is used for a number of different notions in the literature. We have in mind sequential-time algorithms that can exhibit unbounded parallelism but only bounded sequentiality within a single step. Bounded sequentiality means that there is an _a priori_ bound on the lengths of sequences of events within any one step of the algorithm that must occur in a specified order. To distinguish this notion of parallel algorithms, we call such parallel algorithms _wide-step_. Intuitively the width is the amount of parallelism. The ‘step’ in ‘wide-step’ alludes to sequential time.”
Small-Step Characterization Theorem. This is Theorem 6.13 in [10], according to which every noninteractive small-step algorithm is behaviorally identical to some noninteractive small-step abstract state machine. The notion of noninteractive small-step algorithms is defined axiomatically. Noninteractive small-step ASMs are the sequential ASMs of the Lipari guide without the import command. The point of removing the import command is to make these ASMs noninteractive. In a trivial way, the theorem generalizes to algorithms whose interaction with the environment is purely inter-step.

Wide-Step Characterization Theorem. This is Theorem 10.1 in [3] according to which every noninteractive wide-step algorithm is behaviorally identical to some noninteractive wide-step abstract state machine. The notion of noninteractive wide-step algorithms is defined axiomatically. The notion of a noninteractive wide-step ASM is a variant of the Lipari guide notion of parallel ASM. In a trivial way, the theorem generalizes to algorithms whose interaction with the environment is purely inter-step.

3 Ordinary Small-Step Algorithms

This section reflects joint work with Andreas Blass.

What happens if an external function of an ASM “stalls”? The Lipari guide does not address the issue directly but the tradition has been that the ASM is stuck waiting for the environment to deliver a reply to the query. Typically such a behavior is correct. The algorithm may be waiting e.g. for a user to key in necessary information.

However, it does not follow from first principles that, in order to finish a step, an algorithm should wait for the replies to all queries. Consider for example the following algorithm.

Start a step by posing Boolean queries $\alpha$ and $\beta$. If both of them return \texttt{true} then output 7, finish the step and halt. If at least one of them returns \texttt{false} then output 11, finish the step and halt.

Breaking another assumption implicit in the Lipari guide, we can make the above algorithm sensitive to the order in which the replies to $\alpha$ and $\beta$ occur.

Start a step by posing Boolean queries $\alpha$ and $\beta$. If both of them return \texttt{true} and the reply to $\alpha$ precedes the reply to $\beta$ then output 6, finish the step and halt. If both of them return \texttt{true} but the reply to $\alpha$ does not precede the reply to $\beta$ (so that either the reply to $\beta$ precedes the reply to $\alpha$ or else the two replies occur simultaneously) then output 8, finish the step and halt. If at least one of them returns \texttt{false} then output 11, finish the step and halt.

On the other hand, we have yet to see applications violating either of the two implicit assumptions. “Accordingly, we study in this paper”, we quote from [5], “those algorithms, the ordinary ones, that
never complete a step until all queries from that step have been answered
and
use no information from the environment beyond the function assigning an-
swers to queries.

Here we count a time-out signal as an answer. A more explicit formulation of the
second aspect of ordinariness is that whatever the algorithm does is completely
determined by its program, its current state, and the answers the environment
has already provided for earlier queries.”

Ordinary small-step algorithms are axiomatized in [5]. In [6], we generalize
the Small-Step Characterization Theorem to ordinary small-step algorithms.
The ordinary small-step ASMs do not adhere to the Lipari guide completely.
They can nest external functions. And there is something else. The Lipari guide
interprets distinct invocations of the same external function with the same arg-
ments during the same step as the same query.

Think about an external function as a (dynamic) oracle. The [ASM]
provides the arguments and the oracle gives the result. The oracle need
not be consistent and may give different results for the same argument
different times. . .
However, the oracle should be consistent during the execution of any
one step of the program. In an implementation, this may be achieved by
not reiterating the same question during a one-step execution. Ask the
question once and, if necessary, save the result and reuse it.

This interpretation is too narrow. Sometimes the opposite interpretation is ap-
propriate. For example, all invocations of the import command give rise to dis-
tinct queries. (Even though import was not treated as an external function in the
Lipari guide, such treatment is natural.) But neither of these two extreme inter-
pretations is appropriate in all cases. We need to be more flexible and general.
That issue is taken up in [6].

4 General Small-Step Algorithms

This section reflects joint work with Andreas Blass and Benjamin Rossman.
To deal with arbitrary (intra-step interactive) small-step algorithms, we take
advantage of the fact that a small-step algorithm presupposes the existence of a
single agent who is in charge of the algorithm’s execution. This key observation
allowed us to axiomatize general small-step algorithms [7]. The work on the gen-
eralization of the Small-Step Characterization Theorem to intra-step interactive
algorithms is in progress.

5 Wide-Step Algorithms

In order to generalize the Wide-Step Characterization Theorem to intra-step
interactive algorithms, we need to “marry” the advance on intra-step interaction,
sketched above, with the analysis of wide-step algorithms in [3]. Interestingly, the analysis may be simplified in the following aspect. In [3], every proclet (a small-step component of the wide-step process) can update its mailbox at most once during any one step of the whole wide-step process. In the new framework that allows intra-step communication, a proclet can update its mailbox several times during a single step. This gives some technical benefits.

References

Closed-Loop Modeling and Related Problems of Embedded Control Systems in Engineering

Hans-Michael Hanisch
Department of Engineering Science
Martin Luther University Halle-Wittenberg
Kurt-Mothes-Str. 1
06120 Halle, Germany
Hans-Michael.Hanisch@iw.uni-halle.de
http://at.iw.uni-halle.de

Abstract. The contribution tries to present some issues coming from control technology in a way that is related to embedded or reactive systems in computer science and technology. It therefore reviews some of the very basic design patterns of control engineering and shows the differences and similarities of the control engineering approach and methodologies taken from computer science and technology. It focuses on development of closed-loop and model-based design patterns for Programmable Logic Controllers for discrete state/transition or even hybrid systems. Some of the major issues in this field are discussed. Current limitations and obstacles are highlighted, and some ideas about how they could be overcome in future are presented.

1 Introduction

The field of control engineering, especially of designing, manufacturing and programming of controllers, is more and more influenced by computer technology. This is only natural since controllers are a special kind of computation devices. It is, however, only half of the truth in a broader sense that is needed to deal with automation technology.

This contribution is intended to provide a view on controller design in the field of manufacturing and process systems. It tries to explain the role a control engineer plays in the general engineering process and some of the design methodologies that are – or might be – used. Special attention is directed to the closed-loop design pattern for systems that show some discrete state/transition behavior or even hybrid behavior.

A closer look on the problems that need to be solved tries to illustrate limitations of currently available methods and the reasons why application of formal methods to practice is rather sparse.

The contribution is therefore organized as follows.

Section 2 will briefly characterize the current situation that can be observed in the field of control engineering. This section ends with a sketch of a methodology that marks the steps towards future design of logic controllers for discrete state/transition or hybrid systems.
Sections 3 and 4 take a closer look on the major problems of this methodology, namely the transformation of informal or semiformal knowledge to exact models representing this knowledge. These are the most critical issues. One is the design of a model of the object that is to be controlled, and the other one is the formalization of behavioral specifications of that object. The capabilities and the limits of formal methods that use the results of Sections 3 and 4 are discussed in Section 5. In particular, the focus will be directed to requirements that come from the specifics of control engineering and the current problems and limitations that can be observed so far.

The paper will be concluded by some remarks about future development of controller design and will give some ideas for future, interdisciplinary research.

2 Characteristics of Control Engineering

A person taking a first glance of control engineering from outside would probably think of it as a branch of electrical and computer engineering, especially software engineering. Obviously, designing and programming systems for information processing is one of the tasks a control engineer has to accomplish.

The “truth”, however, lies a bit deeper.

All those tasks that are accomplished and control systems that have to be designed and operated are only a means to reach goals that come from outside.

The goals of control are set by the processes and systems that are controlled. In the cases that are of interest here, such systems are manufacturing systems or process systems. Such systems are operated for manufacturing some kind of goods that can be sold and therefore bring in the money for the particular company. This is actually the goal of the whole thing. From this point of view, it is only natural that this goal dominates also the goals of control. Since this is well-known in the engineering community, control engineers learn that the initial point in everything they do is to study the behavior of the controlled object, called “plant”. Despite of a lot of similarities between controller design in engineering and “embedded systems” or “reactive systems” in the computer science and engineering area, the focus is different.

At least in manufacturing and process industry, the plant is in almost all cases unique, pregiven and unchangeable to the control engineer. Things are different in the aerospace industry, automotive industry etc., but this is beyond the scope of this contribution.

So, in contrast to design methodologies for embedded systems where the focus lies on the behavior of the controller having some ideas of the behavior of the environment in mind, the focus in our case is on the behavior of the plant having some ideas of control in mind. This difference has consequences as one can see later on.

An example should illustrate the problem statement.

Figure 1 shows the equipment of a student’s practice plant installed in the former lab of the author at University of Magdeburg. It simulates a typical process for manufacturing of chemical substances but is designed in such a way that no product (and no waste) is actually produced. The upper part forms a
Fig. 1. Example: Batch Process

dilution process of a salt solution. Concentrated salt solution (5 g/l) is stored in the tank on the upper left side. The tank on the upper right side contains pure water. A specific amount of diluted solution (called a “batch”) is produced in a stepwise, sequential process. The tank in the upper middle part is equipped with a stirrer and serves for dilution of the concentrated solution with pure water. In a first step, a defined amount of concentrated salt solution is charged to this mixing tank. After switching the stirrer on, pure water is added until a concentration of 3 g/l is reached. Then the diluted solution is completely discharged into the storage tank below.

The lower part is for separating the water from the concentrated solution to get the ingredients back. An evaporator together with a condenser are provided for this purpose. One batch of diluted solution is filled into the evaporator, the heating is switched on, and the evaporated water is condensed in the condenser and stored in a storage tank at the lower right part. This process continues until a salt concentration of 5 g/l is reached in the evaporator. The concentrated solution is discharged to the storage tank at the lower left part where it is cooled down. The concentrated solution as well as the water can be pumped back to the appropriate storage tanks (upper part) where the substances can be used over and over again.

Note that several batches can be processed concurrently in the plant. One can be diluted while another one is evaporated and a third one is stored in the storage tank above the evaporator. Note also that it is not permitted to mix batches. Each batch forms an individual amount of substance that must be able to be identified as long as the production of it is in process.
One can see that we have a system that has discrete processes together with continuous processes. Hence, it has a hybrid nature that makes it complex. Another source of complexity is the concurrent manufacturing of several batches that requires resource allocation, co-ordination etc.

Taking into account that the above example is in laboratory scale, one could imagine how complicated and large real manufacturing systems might be and how much knowledge is required to understand and specify their behavior in detail.

![Programmable Logic Controller](image)

**Fig. 2.** Programmable Logic Controller

The controllers that have to ensure correct operation of the plant are completely different from the plant. Controllers are programmable devices for information processing. They are physically equipped with inputs and outputs for signals from/to the plant. Figure 2 shows a typical Programmable Logic Controller (abbr.: PLC). Behavior and means for programming PLC’s are currently defined in the International Standard IEC 61131. The execution of the control program is realized in a cyclic, sequential way as shown in Figure 3. The languages for programming as defined in [IEC93] are – from the point of view of software engineering – rather simple and not very powerful. There are some very low-level languages like simple ladder logic diagrams, but also more powerful and high-level languages like structured text.

In summarizing the above example, one can see the following consequences.

The plant behavior dictates the controller design. Incorrect understanding of the behavior of the plant leads to incorrect behavior of the controller. It is current practice that the first version of a control program is never correct. Extensive tests and corrections of the control program are the consequences. If tests are performed by using the real plant, such tests are extremely expensive since material, energy and other resources are wasted. Such tests can even be dangerous and must therefore be limited to a minimum or even to zero. Any methodology that minimizes the costs of testing could pay off significantly. The present practice, however, shows a different picture. Despite of improvements of the programming tools, the general way of programming has not changed. Up to now, logic control programs are written based on (informal, textual) specifications of the desired or forbidden plant behavior without any model of the plant.
In fact, plant and controller always interact in a closed loop as depicted in Figure 4. Sensors in the plant provide information about some (usually not all) process variables of the plant. This information is transmitted via signals to the controller inputs. The output signals of the controller trigger actuators that influence the behavior of the plant.

There is not only one single loop, but there are a lot of feedback loops, and also the human being who operates the plant establishes such loops.

Although this is trivial, it is currently not reflected to the extent it should be in the design process of logic controllers.

Thinking in closed-loop patterns is natural to any control engineer in the field of continuous control.

In the classical control theory, a plant is modeled based on laws of physics, chemistry, thermodynamics etc. First of all, balance equations are written down which are then transformed to a set of differential equations and a set of algebraic equations. The variables usually take their values from a subset of the real
numbers. The controller and the specifications can be also described by equations. Hence, the whole domain of the problems and formalisms lies in the set of real numbers with appropriate well-developed mathematics and control theory.

Things look quite different if only one of the variables which are needed cannot be modeled by differential or algebraic equations but must be represented by some kind of discrete state/event model. The example above shows these characteristics.

Since a closed-loop and model-based approach to the design of logic controllers is a very natural thing, methodologies exist in academia that include modeling of the plant. All those different methodologies follow more or less the design pattern that is shown in Figure 5. The different steps of these methodologies will be subject of further description in the following sections where it will be shown why it is so difficult to realize such a “simple” design pattern.

![Model-based Design Methodology](image)

**Fig. 5.** Model-based Design Methodology

### 3 Plant Modeling

The model of the plant is always the crucial point in all closed-loop approaches. In contrast to continuous control, where models have a sound basis in conservation laws of physics, chemistry, thermodynamics etc., discrete state/transition models have to be “designed”. This means that there is not such a nice, systematic way to come up with models like in continuous control. Discrete state/transition models are usually designed from scratch. This might be acceptable for academic, very small examples to illustrate some theory. It is, however, absolutely unacceptable for real systems. The costs for modeling even for the small
laboratory plant shown above would be dramatically high and would exceed the benefits one could get from applying a model-based design methodology. This is one of the main obstacles that prevent the application of model-based design technologies in practice. Since the models must represent the uncontrolled behavior of the plant, i.e. the complete behavior that is physically possible, they tend to be enormous large and complex. Additionally, there is also some psychological obstacle. Any result of a formalism based on such models is only as good as the model. Hence, the engineers who are using model-based technologies have to “trust” the model. They have to be convinced that the model correctly reflects the aspects of plant behavior they are interested in. And it should be kept in mind that the engineers who are experts in plant behavior are not experts in formal modeling.

As a consequence, a way to engineer models rather than to design them from scratch would be required. It should be as transparent and efficient as possible. It is the vision that the models could come together with the equipment that is used in the real plant as an intellectual property of the vendor of the particular equipment. Generating the model of the whole plant would then be done by composing the models of the single pieces of equipment in an appropriate way. This might be possible in far future for simple systems with a clear structure. Manufacturing systems might be candidates since the single operations they perform are simple. The complexity in such systems comes merely from the size of the system, i.e. from the human being who designs such systems. Application of such methods to process systems is rather unlikely since in process systems the complexity comes from the inherent dynamics of the chemical components and reactions, i.e. from “mother nature”.

Things are getting even worse if hybrid behavior must be modeled as it is almost always the case in process systems. Although hybrid automata [AD94] and some extensions of Petri nets with hybrid behavior [DA92] are good candidates for developing modeling formalisms, means for analysis are very restricted. The underlying dynamic behavior of the plant is in almost all cases nonlinear. Even emptying a tank in the batch plant example presented above is nonlinear! Linear models are therefore in most cases only an approximation, and it must be checked in advance whether this approximation is good enough. There is no general answer to this question.

Models should be graphically and executable. This is needed for discussion within groups of engineering staff and for some validation aspects during modeling. Countless modifications of Petri nets have been developed for solving these problems, and also countless authors have claimed that their work has been applied in engineering over the last few decades. But if one really looks into practice of control engineering, the traces they left are almost invisible. It has also often been claimed that the Sequential Function Charts (abbr.: SFCs) that are defined in IEC 61131 are Petri nets [DA92]. But this is wrong since the execution semantics of SFCs is much more complicated than in classical Petri net. Hence, despite of the graphical representation and some structural properties,
there are not much similarities and no much chance to apply theoretical results of the classical Petri net theory to their modeling and formal analysis.

Obviously, also numerous models based on finite automata have been developed in this area. Since models of the uncontrolled behavior are rather huge, there may be complexity problems if the composition of several models requires the computation of the cross product. State charts [Har87] have recently become popular together with UML [RJB98] and some modeling and simulation tools [ABRW03]. The main question here is the translation of these models to models that provide means for formal analysis/verification or even for controller synthesis.

It is worth being mentioned that there was a special research program funded by the Deutsche Forschungsgemeinschaft in the area of modeling and controller design of hybrid systems. A comprehensive overview of available methods and contributions of the participants of this program can be found in [EFS02].

What kind of model is ever used, it is a model, but it is not an engineering technology for models. This is the missing link. It is obvious that formal models origin in and are strongly influenced by computer science. Any new result regarding modeling power, analysis capabilities etc. can be used for control engineering, and there is really a large field of possible collaboration between both communities. Design and engineering technologies for models of plants, however, must come from the engineering community. Although there are some first steps taken by international initiatives, the way towards this goal is rather long.

4 Specifications

Similar to plant modeling, specification of the behavior of the plant that has to be ensured by control is a challenge to engineers. Specifications are formulated not only by a single person. An engineer responsible for design, construction and erection of the plant will most likely define specifications that are relevant for safe operation of the plant. Another engineer who is responsible for efficient operation of the production system would formulate other specifications for optimal processes in terms of minimization of time and effort. An engineer who has his or her main focus in the field of product design and quality maintenance would specify properties of the raw material needed and the product that is actually manufactured etc.

Formalization of all these different types of specifications is extremely difficult and is an interdisciplinary task with extensive communication and discussion among persons with different technological background and specific knowledge.

It is neither clear whether the set of specifications is complete (in what sense ever) nor non-contradicting. It is rather most likely that specifications are contradicting.

One can observe that specifications form at least three groups, namely:

- Plant specifications,
- Process specifications, and
- Product specifications.
Plant specifications can often be formalized as forbidden state problems, but they might also specify forbidden sequences of states or of state transitions.

Process specifications can be formalized as a set of partially ordered states or state transitions, sometimes even with time or hybrid dynamics. Numerous product specifications cover an extremely wide range. Specifications of substances in process industries define chemical or physical properties of the products, as, for example, in pharmaceutical industry, a very strictly specified amount of substances in a product.

Product specifications in the manufacturing industry focus on geometrical or mechanical properties, color, surface properties etc.

One could hardly think about one unique methodology for formalization that would be able to cover the majority of specifications in this wide range.

One observation, however, is always true: The specifications do not say anything about the controllers. The specifications make statements regarding the plant and the properties of the material that is processed.

It has already been mentioned that specifications may contradict each other. Such contradictions, however, can in general not be detected if one only looks at the specifications themself without knowledge of the plant. The batch plant shown in Figure 1 is a good example. The evaporator is equipped with an electrical heating. The heating must always be covered with solution when it is switched on. If the level of salt solution in the evaporator is lower than a given limit, the surface of the heating is exposed to air and will therefore be overheated and damaged. Hence, such a state of operation is forbidden. On the other hand, the evaporation process must be continued until the desired concentration of salt (5 g/l) in the solution is reached. In usual operation, this is no problem. If, however, a smaller amount of diluted salt solution is charged into the evaporator, the initial level of solution is already low, and if the level is further lowered by evaporating water, a state can be reached in which the above mentioned specifications may contradict. Even a model of the plant that describes its logic behavior is not sufficient to find such cases, but the whole hybrid nature of plant behavior must be modeled to detect such possible contradictions.

In practical control engineering, such a case is a nightmare. It might not be found by extensive tests if always enough diluted solution is available. But it might happen from time to time in daily operation with the result that processes will get stuck, that manual intervention is required and product is lost. One could say that the simplest thing is to add another specification that says that initially the evaporator must always be filled with enough solution to prevent such a case. This may cause another contradiction with the specification that says that batches must not be mixed. So, if the amount of substance that is stored in the storage tank above the evaporator is to small, it is not allowed to be discharged to the evaporator. The process would get stuck a few steps earlier with the same result of manual intervention and loss of product.

Due to the high importance of specifications in engineering and its relevance to research, there is also a special program of the Deutsche Forschungsgemeinschaft that is currently going into its final phase and that focuses on integration
of specification techniques for engineering applications. It has therefore a much broader scope than this contribution and gives a comprehensive view of the vast area of very different problems and methodologies [Ehr04].

5 Controller Synthesis and Verification

Controller synthesis and controller verification are two methods to finally end up with a controller for which the correctness is proven. Both need formal models of the uncontrolled plant behavior, and both need formal models of the specifications. The ways these inputs are used, however, are different.

5.1 Synthesis

Controller synthesis is a formalism that has the formal model of the plant and the formal model of the specification as inputs. The synthesis formalism itself produces a formal model of a controller that guarantees that the specifications would be fulfilled in the closed loop. So, in contrast to verification that has to be done over and over again if the behavior is not correct and needs to be changed, the synthesis procedure has to be performed only once. It is therefore the most elegant way of logic controller design. Means for synthesis, however, are very limited. The limitations are:

1. Very simple and abstract models of plant and specifications are used in currently available methods. Most of them are based on finite state machines and are inspired by the fundamental work of Ramadge and Wonham [RW87]. Others use Petri nets and derivatives. Basic work in this domain was done by Krogh and Holloway [HK90].

2. Synthesis methods exist only for a very limited number of types of specifications. Forbidden state problems are most of them. Synthesis methods based on finite state machine models can also deal with specifications of sequences. The main obstacle here is that process specifications with partial order semantics must be transformed to interleaving semantics.

3. The resulting models of controllers or supervisors are also rather abstract. Additional information is required to implement them on a logic controller. Hence, the implementation can up to now not be performed automatically and must be done manually. This obviously causes significant additional effort and also additional errors.

4. Almost all methods suffer from exponential complexity that comes mostly from the need to compute the whole set of global states of the system.

A major issue is not only safety but also the absence of blocking. If plant and controller would deadlock, the plant would be safe but could not perform any manufacturing process. Additional effort is needed in most of the synthesis procedures to get rid of blocking states.

Synthesis methods that include time behavior or even hybrid one are yet rather sparse [EFS02].
Another issue is that there are no common states and no common state transitions in the plant and in the controller as it is often assumed by formal methodologies for controller synthesis. The interaction must be modeled as it really is, namely only by flow of signals. The use of these signals is usually limited in real systems. Not all of the behavior of the plant can be controlled. Most manufacturing systems have uncontrollable (i.e. autonomous) subprocesses. In particular, the problem becomes even more complicated when the substances have their own dynamic behavior as it is the case in process systems with chemical reactions, heat transfer and many other phenomena. The same problem is how to get information about the current behavior of the controlled object. Not all process variables can be indicated by appropriate sensors. Synthesis algorithms have to provide means to incorporate such properties. This is not always the case.

The controller model that is the result of the synthesis algorithm is not the controller itself. It needs to be implemented on the physical controller with a given performance, cycle time, storage capacity etc. This is actually a problem that does not require any knowledge about the closed-loop behavior. It is a field in which computer technology, especially software engineering, can significantly support future development.

5.2 Controller Verification

Controller verification takes a model of the uncontrolled plant behavior and a model of the controller to compose both to a model of the closed-loop system. This model is subject to verification/falsification of some formal specifications. One should keep in mind that most of the specification in this domain are given in terms of the plant. Open-loop verification of only the controller does not make sense in general although there exist exceptional cases in which this is enough.

Verification has made sincere progress by the research on modelchecking in the field of computer science. From the point of view of control engineering, critical issues remain to be solved.

Transformation of Program Code to Models. This step in the methodology can in principle be completely formalized. The problems herein are the same as in any other modeling methodology for software. Low-level languages may be modeled completely whereas in high-level languages some restrictions could apply. In addition, the programming languages in IEC 61131 are not completely specified. Although IEC 61131 is an international standard, vendors of programmable controllers yet use different variations of the languages in their programming environments.

Composition of the Closed-Loop Model. There are several critical issues. First of all, the plant (and therefore the model of it as well) shows some asynchronous, timed or even hybrid behavior whereas the controller (and its model) has a synchronous behavior. The resulting model of the closed-loop system shows
then a mixed behavior. This means that any type of formal model must be capable to express such mixed behavior. Second, composition of formal models is mainly performed by sharing some common states or some common transitions. If we look to reality, however, there are neither common states nor common transitions among plant and controller. They are interconnected by signals that come from sensors to the inputs of the controllers and go from the outputs of the controller to actuators in the plant. This pattern of interaction establishes some kind of interaction. This means that the plant behavior is not stopped until the controller has read a new value coming from a sensor and that also the controller behavior is not stopped until an actuator has changed its state. The interaction via signals is “one-sided” without backward effects to the system that emits the signal. Formal models that are used must be capable to express such a type of synchronization.

Third, the dynamics of plant and controller is dramatically different. The timescale of state changes in the plants shown in Section 2 is seconds or minutes whereas the controller performs its cyclic behavior in milliseconds. Hence, the timescale of the composed model would be in the range of milliseconds. If it had some discrete clocks, it is obvious that even for very small models the number of states would explode.

Up to now, it takes some “tricks” to overcome this difficulty. To the author’s knowledge, there is yet no general, concise and keen methodology how to handle those problems.

**Debugging.** The term debugging means the iterative process of finding and correcting design errors in the controller. It is rather unrealistic to suppose that the initial design of a controller is correct. This leads to another critical issue, namely the way how the control engineer is involved in this process. There is no doubt that transformation of semi-formal descriptions used in the field of engineering into very basic models of computer science that provide good capabilities for modelchecking is a way to take advantage of the nice results in modelchecking. Any transformation, however, also requires a re-transformation in the case that an error was found and a counterexample is given by the modelchecker. The trajectory of this counterexample has to be presented to the engineer in the description method he uses because he has to fix the error. This means a representation (preferably in a graphical way) in terms of the plant behavior and of the controller behavior as well. This may cause additional problems and in any case additional implementation effort for appropriate software tools to hide the modelchecking engine as much as possible and to integrate all those features in a software environment the engineer is used to apply for controller programming.

### 5.3 Summary

In summarizing this short view on controller synthesis and verification, one could come up with the conclusion that controller verification is much closer to application than controller synthesis. Nonetheless, too much optimism is not justified
because there are not only technical obstacles. One must know that most of these features are not yet taught in regular courses on logic controller programming at all universities, not to mention the “Universities of Applied Sciences”. This has the consequence that even young engineers coming from universities are not or not in detail aware of such techniques and capabilities.

6 Conclusion

Despite of all these problems mentioned above, there must be some work being done in the domain of model-based and closed-loop methodologies. There is no doubt that only such methods are able to improve correctness as well as performance of logically controlled systems. Although applications in daily practice are yet far ahead, one has to think about the research directions that may eventually bring this vision a bit closer to reality. Not to speak about what engineering has to do in its own domain, some remarks about interdisciplinary collaboration in this field seem to be useful.

1. A serious consideration of the problems and mutual understanding among different scientific disciplines is required. The problems are present and will not disappear when they are ignored.
2. A better understanding of capabilities and limitations of formal methods in the application to control engineering is necessary. Unrealistic expectations lead to frustration and create additional obstacles.
3. There is no single model (probably not even a single type of formal model) that would be able to serve all purposes. Means for systematic refinement and abstraction of models are needed. Such means must be part of an engineering process and therefore understandable to and executable by an engineer who is not an expert in formal models.
4. There will be no single research group or a single software tool that could solve all problems. Instead, a smaller focus of different groups would be helpful. This means that models, methods and appropriate software tools should be capable to be combined to create a more complex framework.

Collaboration instead of competition is required more than ever. Both disciplines, control engineering as well as computer science, have their own core fields of expertise that cannot be replaced or substituted by other ones. Even if the above glance is not a very optimistic one, there is no other way than to continue work in this direction.

Acknowledgements

The author gratefully acknowledges – among others – the support of the Deutsche Forschungsgemeinschaft under grant Ha 1886/12-2 and of the National Science Foundation under grant NSF INT-0331796.

Thanks to Thomas Both for his help in preparation of the final manuscript.
References


An ALGOL-View on Turbo ASM

Hans Langmaack

Institut für Informatik und Praktische Mathematik der Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40, D-24098 Kiel, Germany
hl@informatik.uni-kiel.de

Abstract. The transition from basic ASMs to Turbo ASMs reminds at the earlier transition from programming languages like FORTRAN and ALGOL58 to ALGOL60. Essential new features are on the ASM-side naming, parameterizing, local states and possible return values of rules and were on the ALGOL60-side procedures, block concept and function procedures with return values. Turbo ASM–theory and ALGOL60 have two central phenomena in common: Operational style of specifying and semantics definition and named rules and procedures with their potential of recursion. There are quite a few connections between Turbo ASM and ALGOL60. The close coupling of operational style and recursion becomes especially evident in Apt’s and Olderog’s proof of soundness of Hoare’s deduction rule for recursive ASM–rule resp. procedure calls.

The transition from basic ASMs to Turbo ASMs in works like [BoB03, FrS03, BoS03, SSB01] reminds at the earlier transition from programming languages like FORTRAN and ALGOL58 to ALGOL60 [PeS59, SaB59, Sam57, Nau60]. Essential new features are on the ASM-side naming, parameterizing, local states and possible return values of rules and were on the ALGOL60-side procedures, block concept and function procedures with return values. The authors of [BoS03] state: “We extend the basic ASMs by parameterized submachines which may recursively call themselves and thus genuinely enrich the notational macro-shorthand”. The authors of FORTRAN [Bac57] in the nineteen fifties and of ALGOL58 [PeS59] saw their functions resp. procedures in a similar manner only as comfortable shorthand notations for macro-expanded programs without functions and without procedures. This paper shall show up that there are quite a few correspondences between ALGOL60 and Turbo ASM.

1 General Recursive Functions

ALGOL60 was so modern in those days of 1960 that general recursive, partially defined functions in the sense Gödel and Herbrand [Her61] could be immediately expressed by ALGOL60–programs as it can be done by Turbo ASM–specifications. Every such function can be represented by a system of non-nested, mutually recursive function procedure declarations with call by value (i.e. strict) parameter transmission. Let us demonstrate that for multiplication of natural numbers (including zero):
begin
  integer x, y;
  integer procedure add (x, y); value x, y; integer x, y;
  begin add :=
    if y = 0 then x
    else add (x, if y ≤ 0 then 1 ÷ 0 else y−1) +1
  end;
  integer procedure mult (x, y); value x, y; integer x, y;
  begin mult :=
    if y = 0 then 0
    else add (mult (x, if y ≤ 0 then 1 ÷ 0 else y−1), x)
  end;
  read (x, y); comment only natural numbers (including zero) are admissible as inputs. The program terminates successfully and correctly for all such inputs;
  print (mult (x, y))
end

All such programs which calculate general recursive, partially defined functions use only the successor • + 1, partial predecessor if • ≤ 0 then 1 ÷ 0 else • − 1 and equality • = • as standard functions and zero 0 as standard constant, all called static functions in ASM–theory. We use the curious expression to represent the partial predecessor function for natural numbers just to please the syntax of ALGOL60. Partial definedness shows up by erroneous division 1 ÷ 0 or by nontermination due to infinitely repeated callings of user defined function procedures. The latter ones correspond to named and parameterized rules in ASM–theory. The integer ALGOL–variables x and y are examples of 0-ary dynamic function names, the parameters x, y play, beside their role as parameters, the additional role as local let-variables. Turbo ASM’s reserved 0-ary function result is represented by left hand function procedure identifier occurings add := and mult :=. read(x, y); establishes an initial state A₀ of our present Turbo abstract state machine for multiplication of natural numbers, print(mult(x, y)) represents the (here unnamed) main rule which yields an update of the output medium.

2 Procedure Identifiers and Rule Names as Parameters

Rule parameter transmission in Turbo ASM is by name. Call by value is simulated using the let-construct. The parameter transmission in ALGOL60 is essentially also by name, call by value parameters are treated as in Turbo ASM. The call by name concept in ALGOL60 was overtaken from Church’s λ-calculus [Her61,CuF68].
Formal parameters are allowed not only to be variables for actual function terms, they are also permitted to be variables for actual rule names as is demonstrated in [BoS03] e.g. for PRIMITIVERECURSION \((G, H)\). Let us rewrite that so called ASM–rule scheme in ALGOL60 to compute addition on the basis of succession and multiplication on the basis of addition:

```plaintext
integer procedure PRIMITIVERECURSION(x, y, G, H); value x, y;
    integer x, y; integer procedure G, H;
    begin integer ival, rec;
      ival := G(x); rec := 0;
      for rec := rec while rec < y do
        begin ival := H(x, rec, ival); rec := rec + 1 end;
        PrimitiveRecursion := ival
      end;
    integer procedure Gadd1(x); value x; integer x;
      begin Gadd1 := x end;
    integer procedure Hadd1(x, y, ival); value x, y, ival;
      integer x, y, ival;
      begin Hadd1 := ival + 1 end;
    integer procedure add1(x, y); value x, y; integer x, y;
      begin add1 := PrimitiveRecursion(x, y, Gadd1, Hadd1) end;
    integer procedure Gmult1(x); value x; integer x;
      begin Gmult1 := 0 end;
    integer procedure Hmult1(x, y, ival); value x, y, ival;
      integer x, y, ival end;
      begin Hmult1 := add1(ival, x) end;
    integer procedure mult1(x, y); value x, y; integer x, y;
      begin mult1 := PrimitiveRecursion(x, y, Gmult1, Hmult1) end;
```

Inside the integer procedure \(Hmult1\) we could have employed the integer procedure \(add\) as well as \(add1\).

3 ALGOL60’s Block Concept and Operational Semantics

K. Samelson developed and formulated the block concept for ALGOL60 [Nau60]. Blocks may have simple variable and array declarations, procedure declarations and statements. Blocks are special statements and thus they may be nested. So we have local identifiers with a static scoping and visibility concept and with ranges (scopes) and reaches of declaring identifier occurrences as we know that phenomenon from \(\lambda\)-calculus and predicate logic. The reach of such an occurrence
is the subarea of its range (or scope) where it is visible from. It is hidden in the complement area w.r.t. the range.

Whereas simple ALGOL60–variable identifiers relate to 0-ary dynamic ASM-function names, array identifiers of dimension \( n \geq 1 \) relate to \( n \)-ary dynamic function names. Array declarations in ALGOL60 look for example:

\[
\text{real array } A [0 : 5, -3 : y, 1 : x+1];
\]

O. k., indices in ALGOL60 are restricted to integers of bounded intervals, but that does not violate the principal correspondence. As with many features of ALGOL60: They are often more flexible (and efficiently implementable nevertheless) than in many successor languages, called more modern programming languages.

It is important to have a notion of congruent programs. They are differing only by bound renamings of identifiers such that congruent programs and their subphrases have equal semantics. Several industrially used programming languages, established even later than ALGOL60, do not adhere to the virtue of static scoping. They propagate dynamic scoping with the consequence that congruent programs can have different semantics. Dynamic scoping came into play by an incorrect advice of E.W.Dijkstra how to implement recursive procedures in ALGOL60 [Dij60] and by programming errors in J.McCarthy’s interpreters for the functional language LISP1.5 [McC65] which were thought as a definition of LISP1.5’s semantics. J. McCarthy’s intention was to introduce a handy extension of the applied \( \lambda \)-calculus with its pure static scoping concept. But many implementors and users followed the interpreters in a too literal manner, not in the spirit of the \( \lambda \)-calculus.

Let us now look at the procedure calls of ALGOL60 which correspond to the rule calls of Turbo ASM. We especially study the semantics of procedure calls because there is quite an interesting analogy how researchers in ASM-specification see realistic programming language semantics and because there is an illuminating ALGOL–implementors’ view at the Turbo ASM macro–microstep debate [FrS03,BoS03]. The ALGOL60–report states:

“4.7.3. Semantics
A procedure statement serves to invoke (call for) the execution of a procedure body. The effect of this execution will be equivalent to the effect of performing the following operations on the program at the time of execution of the procedure statement:

4.7.3.1. Value Assignment (call by value). [We leave the section out because call by value is reducible to call by name].

4.7.3.2. Name Replacement (call by name). Any formal parameter not quoted in the value list is replaced, throughout the procedure body, by the corresponding actual parameter, after enclosing this latter in parentheses wherever syntactically possible. Possible conflicts between identifiers inserted through this process and
other identifiers already present within the procedure body will be avoided by suitable systematic changes of the formal or local identifiers involved.

This explains how to avoid so called local binding errors when actual parameters, i.e. identifiers or expressions, are substituting applied occurrences of corresponding formal parameters. Without a suitable bound renaming (which is always available) of identifiers locally declared inside the procedure body it can well happen that identifiers inside actual parameters get locally bound inside the body what will yield an unreasonable semantics in the sense of static scoping as we know that from predicate logic and \( \lambda \)-calculus.

\textit{4.7.3.3. Body Replacement and Execution.} Finally the procedure body, modified as above, is inserted in place of the procedure statement and executed. If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body \([a \text{ so called global parameter}]\) the conflicts between the identifiers inserted through this process of body replacement \([\text{copying}]\) and the identifiers whose declarations are valid at the place of the procedure statement or function designator \([\text{function procedure call}]\) will be avoided through suitable systematic changes of the latter identifiers.”

This explains how to avoid so called global binding errors when a modified procedure body substitutes a procedure call. Let us mention that the envisaged procedure call is outside all procedure declarations. So the call is inside the so called main part of the presently expanded program, especially inside a series of nested scopes which are blocks and generated blocks (modified procedure bodies, so called procedure incarnations). If the called procedure is declared in the smallest of these nested scopes then no global binding errors can occur. Unwanted identifier clashes can only occur with identifiers in those scopes disjoint from the procedure declaration. An appropriate bound identifier renaming to avoid global binding errors and so to achieve static scoping is always available.

The ALGOL60–report does not say words about what to do when procedure incarnation execution finishes. Two equivalent positions are thinkable: Either the incarnation is replaced by the old procedure call. Or the incarnation is left standing and execution directly proceeds to the next program point. The second attitude makes sense in the presence of loops or goto statements. But we tend towards the first attitude because it is closer to real implementations by ALGOL60–interpreters or –compilers. So at any time of program execution the initialized and not yet finished procedure incarnations always form a chain of nested incarnations which is pushed down and popped up at the top end, i.e. the chain is treated as a stack. This vision has lead E.W.Dijkstra to extend F.L.Bauer’s and K.Samelson’s runtime stack for expression evaluation in order to implement ALGOL60 with recursion [SaB59,Dij60].

The authors of the ALGOL60–report established (or better say: indicated) the dynamic semantics definition of a wellformed program in an operational small step style just as R.Stärk, J.Schmid and E.Börger do for Java very recently [SSB01]. Program execution is an abstract program counter’s walk through the annotated abstract syntax tree. The associated state is given by the declaring
occurrences of simple and subscripted variables with data values assigned. Intermediate results of expression (term and formula) evaluation are attached to the corresponding operator nodes. Especially, when execution of a function designator (function procedure call) finishes, then the function result is attached to the designator node.

If bound identifier renamings to avoid binding errors are only done for simple variable and array identifiers and formal parameters, but not done for nonformal procedure identifiers then we speak of dynamic scoping semantics [Old81a]. To start out from distinguished programs does not even help avoiding binding errors [GHL67,Kan74]. Distinguished are those programs whose declaring identifier occurrences are denoted by pairwise different identifiers which above that must be different from all free identifiers. The latter ones are standard in case of a wellformed program. For distinguished programs without procedure nestings static and dynamic scope semantics coincide.

For static scoping there is a simple systematic proceeding to avoid binding errors: Make the program distinguished by a bound renaming before resp. after every execution of a procedure call.

There are good reasons why programming languages and even more specification languages should follow static scoping. Dynamic scoping causes unmotivated changes of identifier meanings which are hard to pursue when the user tries to understand a program and to prove it correct. Furtheron: Static scoping semantics is more powerful than dynamic scoping semantics, inspite of the misleading word “dynamic”. This is because all dynamic scoping induced formal execution trees of programs and specifications are regular, whereas static scoping can induce not only regular trees but any irregular ones which recursively computable tree generation can think of [Lan73a,Lan73b,Old81a,Old81b]. ALGOL60-- or 68--programs [Wij69], even of procedure nesting level $\leq 2$ [LiS78], can simulate Turing--machines without any employment of arithmetic or other data, just by transmitting procedure identifiers as parameters [Lan73b].

Our suggestion is to do controlling tasks which are separated from data manipulations by transmitting procedure identifiers and rule names as parameters (*). Clear, then formal procedure and rule calls become an important instrument. Propagators of object oriented programming claim that formal calls are too hard to understand [Nyg02]. On the other hand W.Goerigk and C.Blaue are working on systematic methods to translate ALGOL-- or Pascal--programs, including formal procedure and function calls, to structurally equivalent object oriented Java--programs which, due to this structure equivalence, cannot be understood better.

The following short ALGOL60--program [Kan74] has just two nested procedures and has, in spite of recursion of procedure $p$, only a finite formal execution tree (formal call tree) with incompatible static resp. dynamic scope semantics. The program is a simplified version of an example in [GHL67]:

An ALGOL-View on Turbo ASM 25
begin integer $x$;
  procedure $p(y)$; procedure $y$;
  begin integer $a$;
    procedure $h(z)$; procedure $z$;
    begin print $(a)$ end $h$;
    $x := a := x+1$;
    $y(h)$ end $p$;
  end
  $x := 1$; $p(p)$ end program
/ \
begin integer $a'$;
  procedure $h'(z')$; procedure $z'$;
  begin print $(a')$ end $h'$;
  $x := a' := x + 1$;
  $p(h')$ end first incarnation of $p$
/ \begin integer $a''$;
  procedure $h''(z'')$ precedure $z''$;
  begin print $(a'')$ end $h''$;
  $x := a'' := x + 1$;
  $h''(h'')$ end second incarnation of $p$
/ \begin print $(a')$ end incarnation of $h$ resp. $h'$

Static scoping leads to a printed result $a' = 2$ because $h'$ does not refer to the most recent incarnation $h''$ of procedure declaration $h$ in spite of E.W.Dijkstra’s claim. His claim is equivalent to doing dynamic scoping, i.e. not doing bound renaming of nonformal procedure identifiers. So after dropping the renaming primes $'$,$''$ in $h'$,$h''$ the actual parameter $h$ in $p(h)$ points to the second instance of procedure declaration $h$, but changes its meaning, because after substitution of formal parameter $h$ points to the third instance. So $a'' = 3$ is printed. E.W.Dijkstra’s incorrect claim is still vivid in modern books on compiler construction [WiM92]. An observation: That our example program has a recursively called procedure $p$ without any conditional statement and is successfully terminating nevertheless, this is only due to the fact that formal procedure calls are allowed. Compare our remark (*) above.

4 Turbo ASM’s Macro- and Microsteps and Their Analoga in ALGOL

N.G.Fruja and R.F.Stärk do an interesting investigation [FrS03] on the hidden computation steps of Turbo ASMs with the help of PAR/SEQ–trees. So Turbo ASM has not only a natural bigstep operational semantics, defined by the yields–relation, there is also an equivalent structural smallstep operational semantics [Plo81].
ALGOL60–programs correspond to purely sequential Turbo ASMs. Therefore such program should have PAR/SEQ–trees as well. How can we determine them? They are essentially tail constituents of runtime stacks of nested procedure incarnations. Let us demonstrate that with the program example of section 3. Assume the program counter is pointing to the front of \( x := a'' := x + 1 \); inside the second incarnation of procedure \( p \). The program counter splits the stack in a head and a tail constituent. The latter one is the following path through the stack:

```plaintext
end program

end first incarnation of \( p \)

\( x := a'' := x + 1 ; \)

\( h'(h'') \) end second incarnation of \( p \)
```

If we had no renamings or we were allowed to drop renamings we could easily read off the PAR/SEQ–tree

\[
\begin{align*}
\text{PAR} & \cdot \{(x,1)\} \\
\downarrow & \\
\text{PAR} & \cdot \{(a,2)\} \\
\downarrow & \\
\text{PAR} & \cdot \{(x,2)\} \\
\downarrow & \\
(a := x + 1 \textbf{ seq } x := a \textbf{ seq } h(h), \mathfrak{A}, \phi)
\end{align*}
\]

But to consider both \( x, p \) and \( a, h \) as global is no solution because that would lead to a wrong printed result 3. The values of \( x \), \( a \) and \( \text{outmedium} \) in state \( \mathfrak{A} \) are 2 , 2 and \( \text{undef} \).

As in ALGOL60 [GHL67,Lan73a,Old81a,Old81b] we need appropriate means in Turbo ASM to deal with local function names and local named rules. Let us add an explicit tenth rule in section 2.4.3 “Transition Rules and Runs of ASMs” [BoS03]. We do so in the spirit of section “Turbo ASMs with local rules” in section 4.1.2 “Submachines and Recursion (Encapsulation and Hiding)”: 

**Local Rule:** \( \textbf{local } F \textbf{ local } E \ Q \)

Syntactic condition: \( F \) is a finite list of local function names \( f_i \), pairwise different, and \( E \) is a finite list of local named and parameterized rules

\[
r_j(x_{j1}, ..., x_{jn_j}) = P_j
\]

with pairwise different names \( r_j \). The scope of \( f_i \) and \( r_j \) is the union area of all \( P_j \) and \( Q \). The scope of \( x_{j
\nu} \) is \( P_j \). Bound renamings can be done with local names in case of undesired name clashes (as with variables in the \( \text{Let, Forall and Choose Rule} \) and with formal parameter variables).

Meaning: Assign the value `undef` to all locations of \( f_i \) in \( F \) and execute \( Q \).
The semantics defining relation \( \text{yields} \ (P, \ A, \ \zeta, \ U) \) of Table 2.2 in [BoS03] must be popped up by "local constituents" because rules \( P \), global states \( A \), variable assignments \( \zeta \) and update sets \( U \) refer to the fixed vocabulary of static and dynamic functions and to the fixed global rule names of the given ASM. \( P \) is now replaced by rule units \( E \mid P \) with an environmental list \( E \) of differently named and parameterized local rules, named different from global rules also. \( A \) is replaced by state units \( A \mid B \) with a local state \( B \) such that the vocabularies of \( A \) and \( B \) are disjoint. Update set \( U \) is allowed to refer to \( A \) and \( B \), and variables in \( E \mid P \) may refer to variable assignment \( \zeta \) whereas variables in global rules do not refer to \( \zeta \), they have no free variables.

The \textit{Local Rule} semantics is as follows:

\[
\text{yields}(E' \mid P, \ A \mid B', \ \zeta, \ U') \\
\text{yields}(E \mid \text{local } F \ \text{local } E \mid P, \ A \mid B, \ \zeta, \ U)
\]

- where \( E' \) is the union of \( E \) and \( \overline{E} \), provided all global rule names and local rule names in \( E \) and \( \overline{E} \) are pairwise different,
- where the vocabulary of \( B' \) is the union of \( F \) and that of \( B \), and the locations of \( F \) in \( B' \) are updated by \textit{undef}, provided \( F \) and the vocabularies of \( A \) and \( B \) are disjoint,
- where \( U \) is \( U' \) minus the updates in \( F \).

This rule is in harmony with Definition 4.1.5 (Turbo ASMs with local functions) in [BoS03]. Bound renamings to achieve disjointness are always available.

The appropriately modified PAR/SEQ–tree looks as follows:

\[
\text{PAR} \cdot \{(x, 1)\} \\
\downarrow \\
\text{PAR} \cdot \{(a', 2)\} \\
\downarrow \\
\text{PAR} \cdot \{(x, 2)\} \\
\downarrow \\
(h'(z') = \text{print}(a') \\
h''(z'') = \text{print}(a'') \mid \\
a'' := x + 1 \text{ seq } x := a'' \text{ seq } h''(h''), A \mid B, \phi)
\]

The “dynamic” part of the vocabulary of \( A \) is \( x \) and \textit{outmedium} and of \( B \) is \( a' \) and \( a'' \), and the momentary values are 2, \textit{undef}, 2 and \textit{undef}.

5 A Connecting View of Operational and Denotational Semantics

Let us look for a connecting view of operational and denotational semantics of ALGOL–programs and of Turbo ASMs. The denotational style to define seman-
tics of a given program or machine $M$ with recursive procedures or rules is to create a continuous functional $\Phi$ such that its least fixpoint $\mu \Phi$ is the essence of the semantics of $M$. Due to the fixpoint theorem [Sco70,MiS76,Bak80] the equation

$$\mu \Phi = \bigsqcup_{\nu \in \mathbb{N}_0} \Phi^{\nu}(\bot)$$

holds, and this is the first step towards our goal.

In case we have an ALGOL60–program with just one parameterless procedure $p$ without local declarations

\begin{verbatim}
begin integer $x_1,\ldots,x_n$ ;
procedure $p$ ;
  begin $\ldots$; $p$ ; $\ldots$; $p$ ; $\ldots$ end ;
  read $(x_1,\ldots,x_n)$ ; $p$ ; print $(x_1,\ldots,x_n)$
end
\end{verbatim}

then this procedure is defining a functional $\Phi$ on state transformations: Interpret $p$ as an arbitrary argument state transformation. Then

$$Sem (\textbf{begin} \ldots ; p ; \ldots ; p ; \ldots \textbf{end})$$

might result in a uniquely determined state transformation. But there might also come up no result, especially in case $p$ is interpreted as the totally undefined state transformation $\bot$ (like \texttt{abort}). $Sem$ means the semantics of ALGOL60–programs without procedures, which we assume to be established in whatever style.

There is a syntactical representation for $\Phi^{\nu}(\bot)$: Firstly apply the copying process to procedure call $p$ $\nu$–times in a simultaneous parallel manner; secondly do a final substitution of all procedure call occurrences $p$ by \texttt{abort}.

Question: Do we have such a close connection between denotational and operational style also in the general case of programs and machines? Let us confine to deterministic machines without parallel executions as the situation is with ALGOL60–programs.

For any given Turbo ASM $M$ it is not sufficing to look at just the call of the distinguished rule of $M$. We must take into consideration all possible calls. So we consider \textit{yields}–quadruples

$$(E \mid r(t_1,\ldots,t_n),\mathfrak{A} \mid \mathfrak{B},\zeta,U)$$

with the fixed global vocabulary $\Sigma_A$ for $\mathfrak{A}$ and the vocabularies $\Sigma_B$ and $\Sigma_z$ for $\mathfrak{B}$ and $\zeta$ which contain the free function names and variables in $E \mid r(t_1,\ldots,t_n)$ not in $\Sigma_A$. We may assume w.l.o.g. that rule units $E \mid r(t_1,\ldots,t_n)$ together with the fixed rule list $E_M$ of the machine are distinguished. We call
the index $i$ of the \textit{yields}–quadrupel, and we require its wellformedness, i.e. especially actual and formal parameter numbers agree for every nonformal rule call.

Let $\text{Ind}$ be the set of all such indices and $\text{Yiq}$ the set of deterministic collections of associated \textit{yields}–quadruples (state–update functions). We say that an assignment of all indices to associated state–update functions is an interpretation $J$ of all possible rule calls outside all rule bodies:

\[
J : \text{Ind} \rightarrow \text{Yiq}.
\]

We have to define a functional

\[
\Phi : (\text{Ind} \xrightarrow{J} \text{Yiq}) \rightarrow (\text{Ind} \xrightarrow{J} \text{Yiq})
\]

by executing rulebodies nonrepeatedly, solitary, as we have done for procedure example $p$. Let an interpretation $\tilde{J}$ and an index

\[
i = (E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z)
\]

be given. We have to determine $J(i) \in \text{Yiq}$. $r$ is declared globally in $E_M$ or locally in $E$:

\[
r(x_1, \ldots, x_n) = Q.
\]

The actual and formal parameter numbers agree, otherwise we would not have considered index $i$ as a wellformed index. The names in $t_1, \ldots, t_n$ are global functions in $\Sigma_A$, functions in $\Sigma_B$, variables in $\Sigma_z$, global rule names in $E_M$ or rule names in $E$. We apply the copy rule:

\[
E \mid Q \overset{\pi_1}{\cdots} \overset{\pi_n}{\cdots}.
\]

Name clashes do not occur due to the assumed distinctness. But we must be aware that the substituted $Q$ might become not fully wellformed: Either the rule name position in a formal rule call is filled not by a rule name in $E_M$ with agreeing actual-formal parameter numbers or a formal parameter in a function term is filled not by an appropriate function term resp. variable in $\Sigma_z$ (The \textit{Call Rule} in Table 2.2 must be subjected to the same precautions in order to achieve full operational–denotational corresponding). Then we define $J(i)$ to be the empty state–update function. Otherwise, by a bound renaming in the substituted body $Q_{\text{subst}}$, we may assume distinctness for $E_M E \mid Q_{\text{subst}}$ also. The vocabularies $\Sigma_A, \Sigma_B$ and $\Sigma_z$ stay invariant. We eraze all local declarations in $E \mid Q_{\text{subst}} , E$ included. The remaining reduced body $Q_{\text{subst}}^{red}$ is perhaps no longer a proper rule because there might occur rule calls without bindings. But that does not matter, we have appropriate interpretations for all rule calls available in $\tilde{J}$ so that $Q_{\text{subst}}^{red}$ can be evaluated nevertheless.

Let us consider any rule call

\[
\tilde{r}(\tilde{t}_1, \ldots, \tilde{t}_n)
\]
in $Q^\text{red}_{\text{sub}}$ which occurs in $E \mid Q_{\text{sub}}$ as well. We collect for $\tilde{r}(\tilde{t}_1, \ldots, \tilde{t}_n)$ all visible local rule declarations and get an extension $\tilde{E}$ of $E$. We collect all visible locally declared function names and local let-variables and get extensions $\tilde{\Sigma}_B$ of $\Sigma_B$ and $\tilde{\Sigma}_z$ of $\Sigma_z$ as well.

$$(\tilde{E} \mid \tilde{r}(\tilde{t}_1, \ldots, \tilde{t}_n), \tilde{\Sigma}_A \mid \tilde{\Sigma}_B, \tilde{\Sigma}_z)$$

is a new index $\tilde{i}$ associated to call $\tilde{r}(\tilde{t}_1, \ldots, \tilde{t}_n)$ in $Q^\text{red}_{\text{sub}}$. We interpret those calls by $J(\tilde{i})$ and evaluate $Q^\text{red}_{\text{sub}}$ in one or the other established way. One way would be to add axioms to the modified Table 2.2 in [BoS03]:

$$\text{yields}(|Q^\text{red}_{\text{sub}}, \mathfrak{A} | \mathfrak{B}, \zeta, U)$$

- where $(\tilde{E} \mid \tilde{r}(\tilde{t}_1, \ldots, \tilde{t}_n), \tilde{\mathfrak{A}} | \tilde{\mathfrak{B}}, \tilde{\zeta}, \tilde{U})$ is in $\tilde{J}(\tilde{i})$.

The collection of all quadruples

$$\text{yields}(|Q^\text{red}_{\text{sub}}, \mathfrak{A} | \mathfrak{B}, \zeta, U)$$

which we can derive represent the evaluated $Q^\text{red}_{\text{sub}}$. We determine the collection (state–update function) $J(i)$ by replacing $|Q^\text{red}_{\text{sub}}$.

$$J(i) := \{(E \mid r(t_1, \ldots, t_n), \mathfrak{A} \mid \mathfrak{B}, \zeta, U) : \text{yields}(|Q^\text{red}_{\text{sub}}, \mathfrak{A} \mid \mathfrak{B}, \zeta, U) \text{is derivable}\}.$$ 

After definition of functional $\Phi$ is established we can show that $\Phi$ is monotonous and continuous. We want to express the semantics of the given Turbo ASM $M$, i.e. of the call of its distinguished rule and of any call $E \mid r(t_1, \ldots, t_n)$ in any starting state $\mathfrak{A} \mid \mathfrak{B}$ and variable assignment $\zeta$, by the help of $\Phi$. The collection of all tuples

$$\text{yields}(E \mid r(t_1, \ldots, t_n), \mathfrak{A} \mid \mathfrak{B}, \zeta, U),$$

resp. written in equivalent notation

$$[E \mid r(t_1, \ldots, t_n)]^{\mathfrak{A}(\mathfrak{M})}_{\zeta} \triangleright U,$$

derivable from Table 2.2 in [BoS03] represents the state–update function

$$[E \mid r(t_1, \ldots, t_n)]^\bullet$$

which rule call $E \mid r(t_1, \ldots, t_n)$ is meaning operationally. We would like to indicate a proof that this function can be expressed

$$[E \mid r(t_1, \ldots, t_n)]^\bullet = \mu \Phi \left( (E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z) \right). \quad (*)$$

The proof goes via

$$\mu \Phi = \bigcup_{\nu \in \mathbb{N}_0} \Phi^\nu(\bot)$$
where \( \perp \) is that interpretation which assigns the empty state–update function to every index. Due to the defining construction of \( \Phi \) it is comprehensible that

\[
(E \mid r(t_1, \ldots, t_n), \mathfrak{A} \mid \mathfrak{B}, \zeta, U) \in \Phi^\nu(\perp)((E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z))
\]

implies

\[
yields(E \mid r(t_1, \ldots, t_n), \mathfrak{A} \mid \mathfrak{B}, \zeta, U) \text{ is derivable ,}
\]

and in case the latter is holding there exists a \( \nu \in \mathbb{N}_0 \) with the former property. This proves the desired equation (\( * \)).

But the connection between denotational and operational semantics is even closer. Let us form the formal execution tree (or a congruent version)

\[
C^\nu_{\text{stat}}(\text{local } E \mid r(t_1, \ldots, t_n))
\]

up to level \( \nu \). This means: Apply the copying process \( C^\text{stat} \) in a simultaneous parallel manner \( \nu \)-times to all rule calls outside any rule body. In a next step reduce that tree by replacing all rule calls (outside any rule body) by \texttt{abort}. It is clear that all rules in that tree

\[
\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid r(t_1, \ldots, t_n)))
\]

are irrelevant now and could be erased. Again due to the defining construction of \( \Phi \) it is comprehensible that

\[
(E \mid r(t_1, \ldots, t_n), \mathfrak{A} \mid \mathfrak{B}, \zeta, U) \in \Phi^\nu(\perp)((E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z))
\]

if and only if

\[
yields(\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid r(t_1, \ldots, t_n))), \mathfrak{A} \mid \mathfrak{B}, \zeta, U) \text{ is derivable.}
\]

So we have the following equations for the approximating semantics

\[
\begin{align*}
\llbracket E \mid r(t_1, \ldots, t_n) \rrbracket^\nu_{\phi} &= \Phi^\nu(\perp)((E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z)) \\
&= [\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid r(t_1, \ldots, t_n)))]
\end{align*}
\]

for all \( \nu \in \mathbb{N}_0 \) and for the full semantics:

\[
\begin{align*}
\llbracket E \mid r(t_1, \ldots, t_n) \rrbracket &= \mu\Phi((E \mid r(t_1, \ldots, t_n), \Sigma_A \mid \Sigma_B, \Sigma_z)) \\
&= \bigcup_{\nu \in \mathbb{N}_0} [\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid r(t_1, \ldots, t_n)))]
\end{align*}
\]

Semantics of rule calls may be readily extended towards rules \( Q \) or even rule units \( E \mid Q \) which are not just rule calls:

\[
\begin{align*}
\llbracket E \mid Q \rrbracket^\nu_{\phi} &= \Phi^{\nu+1}((E \mid r_Q = Q, \Sigma_A \mid \Sigma_B, \Sigma_z)) \\
&= [\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid Q))]
\end{align*}
\]

\[
\begin{align*}
\llbracket E \mid Q \rrbracket &= \mu\Phi((E \mid r_Q = Q, \Sigma_A \mid \Sigma_B, \Sigma_z)) \\
&= \bigcup_{\nu \in \mathbb{N}_0} [\mathcal{R}(C^\nu_{\text{stat}}(\text{local } E \mid Q))]
\end{align*}
\]
where $r_Q$ is a newly chosen rule name. The superscript $stat$ reminds of the static scope copying process dating back to the ALGOL60–report. It should be checked whether ParBlock Rule and Forall Rule can be included in the considerations above.

Around the 1970–years several programming language and compiler construction researchers have used the operational approximating semantics via $\mathcal{R} \circ C^{stat}_\nu$ in order to prove and decide properties of ALGOL–like programs and to study correct working of compilers and runtime systems. See [GHL67, Lan73a, Old79, Har79, Old81a, Apt79] and others. In order to prove program proof rules correct approximating semantics is definitely necessary as we shall see in the next section 6.

6 On Verification of ALGOL-Like Programs and Turbo ASMs. Concluding Remarks

Development of proof calculi for ALGOL–like programs has begun with C.A.R. Hoare [Hoa69] basing on ideas which A.Turing [Tur50] and R. Floyd [Flo67] had before for flowchart programs. 1971 C.A.R. Hoare [Hoa71] included in his deduction calculus a proof rule for recursive procedures which we, in the light of Turbo ASMs, would write

$$\lambda \{ \psi \} E \mid r(t_1, \ldots, t_n) \{ \varphi \}$$

- where $r(x_1, \ldots, x_n) = Q$ is a named and parameterized rule in $E_M E$ and substitution leads to a wellformed rule unit.

The horizontal line separates a premis together with its so called assumptions set from the conclusion below the line. C.A.R.Hoare studies special correctness formulas (triples) $\{ \psi \} E \mid Q \{ \varphi \}$ where $\psi$ and $\varphi$ are pure or first order formulas which in ASM–logic would be written $\psi \rightarrow [E \mid Q] \varphi$. ASM–logic is close to dynamic logic [Har79] and algorithmic logic [Sal70,MiS87].

In our understanding two phenomena are central in Turbo ASM–theory: the operational style of specifying and named rules with their potential of recursion. Both phenomena are closely tied together what shows up in soundness proofs for Hoare’s deduction calculus for correctness formulas $\{ \psi \} E \mid Q \{ \varphi \}$ as well as it has shown up in the past w.r.t. ALGOL–like programs with procedures.

A clearly arranged soundness proof for a deduction calculus goes via soundness proofs for all deduction rules with the help of a lemma: If all rules are sound then the calculus is sound. It was by no means trivial to find an appropriate definition of rule soundness. So E.M.Clarke showed soundness of Hoare’s calculus directly. Clarke had not yet available an appropriate notion of soundness of the
deduction rule for recursive calls. But his reasoning is difficult to comprehend. 1979, during an Oberwolfach conference, K.Apt and E.R.Olderog independently came up with the same idea how to define soundness of deduction rules with the explicit help of approximate semantics.

The usual way to define soundness of a proof rule with assumption sets would be (in case of Hoare’s rule):

For all sets \( \Psi \) of (anteceding in the sense of Gentzen) correctness formulas:

\[ (*) \Psi \models \{ \psi \} E \mid Q_{\frac{1}{x_1}} \ldots \frac{m}{x_n} \{ \varphi \} \] implies

\[ \Psi \\setminus \{ \{ \psi \} E \mid r(x_1, \ldots, x_n) \{ \varphi \} \} \models \{ \psi \} E \mid r(x_1, \ldots, x_n) \{ \varphi \} . \]

Logical consequence \( \models \) is based on validity (partial correctness) of triples \( \{ \psi' \} E' \mid P' \{ \varphi' \} \), namely

\[ [E' \mid P']_\Psi (\tilde{\psi'}) \subseteq \varphi' \]

where the so called range \( \tilde{\varphi}'' \) of any formula \( \varphi'' \) is defined as

\[ \tilde{\varphi}'' := \{ (A \mid B, \zeta) : [\varphi'']^\zeta_B = \text{true} \}. \]

\( A, B, \zeta \) are states and variable assignments with vocabularies \( \Sigma_A, \Sigma_B, \Sigma_\zeta \) (everything considered w.r.t. a given machine \( M \)).

If the above implication \((*)\) would really hold then we could take the following correctness formula

\[ cf := \{ \text{true} \} r = \text{skip} \mid r \{ \text{false} \} \]

and set of antecedents

\[ \Psi := \{ cf \} . \]

Then the premis of \((*)\) is a tautology and the following inclusion holds

\[ \text{true} = [r = \text{skip} \mid r]_\Psi (\text{true}) \subseteq \text{false} = \phi \]

which is a contradiction.

A right definition of proof rule soundness considers logical consequences \( \models' \) which depend on approximate semantics. We have to say:

For all sets \( \Psi \) of anteceding correctness formulas:

(\text{for all } \nu \in \text{IN}_0 : \Psi \models' \nu \{ \varphi \} E \mid Q_{\frac{1}{x_1}} \ldots \frac{m}{x_n} \{ \varphi \} \) implies

(\text{for all } \nu \in \text{IN}_0 : \Psi \setminus \{ \{ \psi \} E \mid r(x_1, \ldots, x_n) \{ \varphi \} \} \models' \nu \{ \psi \} E \mid r(x_1, \ldots, x_n) \{ \varphi \} ).

The logical consequences \( \models' \) are based on the following validity (partial correctness) of triples \( \{ \psi' \} E' \mid P' \{ \varphi' \} :\)

\[ [E' \mid P']_\nu (\tilde{\psi'}) = [\mathcal{R}(\mathcal{C}^{\text{stat}}_\nu (\text{local } E' P'))]_\nu (\tilde{\psi'}) \subseteq \varphi' \]
Apt’s and Olderog’s inductive soundness proof of Hoare’s deduction rule for recursive calls is a highlight in any course on program verification. When A. Poetzsch–Heffter and P.Müller [PHM99] proved Hoare’s calculus correct by the help of the HOL–prover they reinvented Apt’s and Olderog’s soundness definition.

Hoare’s deduction calculus is not complete for ALGOL60, not even relatively complete in the sense of Cook [Cla79,LaO80]. The calculus is relatively complete for exactly those sublanguages of ALGOL60 whose programs have regular formal call trees [Old81a]. They are those programs whose procedures can be denested by the method of accompanying parameters [Lan73b,Old81b]. One might think that a rigorous typing discipline of formal procedure identifiers by finite procedure modes in the style of ALGOL68 or Pascal helps, but that is not true [Cla79]. But in case such programs have at most simple side-effects, i.e. no assignments to variables declared in enclosing, outer procedures, then necessary criteria for existence of sound and relatively complete deduction calculi are fulfilled [Lip77,Cla79,Lan82]. And actually, there are such calculi, provided higher order variables for relations are permitted [Old84,Lan85].

It would be very illuminating to investigate how completeness results of the kind above and those from algorithmic logic [MiS87] transfer to ASM–logic. Parallel and concurrent executions are not yet integrated in the above mentioned completeness results. Research should strive for an integration, especially for a comparison with so called MAX– and liberal semantics and their treatments in algorithmic logic.

The author would like to thank E.Börger and W.Zimmermann for the invitation to present this lecture at ASM 2004 and for many fruitful discussions on different styles how computable functions were understood in the 1930s , on how those different styles are still alive in modern programming and specification languages, and on the merits of operational semantics and its intelligible and provably correct implementation. My special heartly thanks go to Annemarie Langmaack who did the typesetting of this article.

References


An ASM Specification of C# Threads and the .NET Memory Model

Robert F. Stärk\textsuperscript{1} and Egon Börger\textsuperscript{2}

\textsuperscript{1} Computer Science Department, ETH Zürich
staerk@inf.ethz.ch
\textsuperscript{2} Dipartimento di Informatica, Università di Pisa
boerger@di.unipi.it

Abstract. We present a high-level ASM model of C# threads and the .NET memory model. We focus on purely managed, fully portable threading features of C#. The sequential model interleaves the computation steps of the currently running threads and is suitable for uniprocessors. The parallel model addresses problems of true concurrency on multiprocessor systems. The models provide a sound basis for the development of multi-threaded applications in C#. The thread and memory models complete the abstract operational semantics of C# in [2].

1 Introduction

Modern object-oriented programming languages like Java or C# support multi-threaded programming. They allow several threads to run concurrently sharing objects on the heap in the same address space. Each thread has its own frame stack, program counter, local variables and registers. The languages have special syntactical constructs for synchronization. Java has a \texttt{synchronized} statement and \texttt{synchronized} methods, while C# has a \texttt{lock} statement and several attributes that can be applied to classes and methods to control their run-time synchronization behavior.

Although the C# programming languages supports multi-threaded programming directly via special syntax, the underlying thread model is poorly documented and still considered to be part of the library. The Ecma standards for C# [4] and the Common Language Infrastructure [5] contain only a few paragraphs about threads. For example, the \texttt{lock} statement is defined in [4, §15.22] by a reduction to the library functions \texttt{Monitor.Enter} and \texttt{Monitor.Exit} which are not further specified there. Important issues, such as the order of writes to volatile and non-volatile fields, are just briefly mentioned in two paragraphs in [4, §10.10, §17.4.3]. Hence, a program developer has to rely solely on the class library documentation that comes with Microsoft’s .NET framework Software Development Kit [11]. Unfortunately, that documentation is not very precise with respect to threads, locks and memory issues. Moreover, it is not identical with the (XML) specification of the types that comprise the standard libraries in [5, Partition IV, Profiles and Libraries]. For example, specifications
of `Thread.Interrupt`, `Thread.Suspend` and `Thread.Resume` are not included in [5].

If a programmer cannot rely on a simple and precise thread model, the task of writing reliable multi-threaded applications that are correctly synchronized and free of data races and deadlocks becomes very difficult and tedious. Multi-threaded programs depend on the scheduling policy of underlying run-time system and therefore synchronization errors are difficult to reproduce and to debug. Moreover, certain problems may only occur under heavy threading stress in production environments like web services which cannot be simulated during the development cycle. Tools that statically analyze multi-threaded programs for synchronization problems are in general neither sound nor complete. Nevertheless, in some cases the may report a high percentage of all possible conflicts (see [19]).

The Java Language Specification [7, Ch. 17] devotes a whole chapter to threads and locks. However, that specification has been found to be hard to understand and has subtle, often unintended, implications. Therefore, the Java community has proposed a new specification of the semantics of threads and locks often referred to as the New Java Memory Model [10]. Whether the new specification is easier to understand may be doubted. It justifies at least most of the common compiler optimizations which were prohibited by the old one. For a comparison and analysis of the different proposals we refer to [1].

The specification of threads in this article extends the modular definition of the semantics of C# in [2] by a new module C#-T for multi-threaded C#. We focus on purely managed, fully portable threading features of C# and the .NET common language runtime. We do not consider the .NET equivalents of Win32 threading primitives such as `WaitHandle` and their derived classes. We also do not model asynchronous delegates and synchronization domains. The starting point of our model has been the thread model for Java in [17]. That model however is only correct for uniprocessor systems and does not address problems of true concurrency.

For basic terminology on Abstract State Machines we refer the reader to [3,8].

## 2 Threads in Microsoft’s .NET Framework

The thread related features of C# are collected in the `System.Threading` namespace (see Fig. 1). The namespace contains the delegate type `ThreadStart` that denotes the type of functions with zero arguments and return type `void`. The most important classes of the namespace are the `Thread` and `Monitor` classes. Several thread related exception classes derived from `SystemException` are also declared in the namespace.

A thread can be in one or more states of the `ThreadState` enumeration (listed in Fig. 1). Unfortunately, the documentation does not state clearly which combinations of states are allowed for a thread and which are not. Moreover, some of the states are not real execution states of a thread but just boolean flags. The `Background` state, for example, tells the run-time system that it can kill the
namespace System.Threading {
    delegate void ThreadStart();
    enum ThreadState { ...
    ... sealed class Thread {...
    sealed class Monitor { ...
    ... class ThreadStateException {...}
    class ThreadAbortException {...}
    class ThreadInterruptedException {...}
    class SynchronizationLockException {...}

} }  

Fig. 1. The System.Threading namespace and the ThreadState enumeration.

thread and exit when all non-background threads have terminated (similar to the Daemon property of threads in Java). Other states, like StopRequested, are for internal use only and should not be exposed to the programmer in a public enumeration. The Aborted state has a rather obscure meaning (see below). If there is an AbortRequested, why is there no InterruptRequested?

The ThreadState property of the Thread class returns a snapshot containing the states of a thread as a bitset. This information, however, cannot be used for synchronization purposes, since it may already been outdated when it is obtained. Therefore, we do not model the ThreadState property below and use a different set of execution states in our model.

Threads are represented in C# by instances of class Thread in Fig. 2. Unlike in Java, this class is sealed (final in Java terminology) and cannot be subclassed. The constructor of the class takes a pointer to a ThreadStart function which will be executed when the new thread is started. The two static methods of the class, Sleep and ResetAbort, are implicitly called on the current thread.

The constructor of class Monitor in Fig. 2 is private, which means that no instances of this class can be created. The reason is, that in C# (like in Java) every object reference can be used as a monitor and therefore there is no need to create special monitors. The Monitor class contains only static methods. Its Wait, Pulse and PulseAll methods are similar to Java's wait, notify and notifyAll methods of class java.lang.Object.

The Enter and Exit methods of the Monitor class are used to syntactically reduce the lock statement of C# (where o is a fresh local variable):

\[
\text{lock (exp) stm} \implies \{ \text{object o = exp; }
\text{Monitor.Enter(o); }
\text{try \{ stm \}
\text{finally \{ Monitor.Exit(o); \}}
\}
\]

Unlike in Java, the Monitor.Enter and Monitor.Exit methods can be called explicitly in C# programs and hence C# cannot guarantee that a thread holds no more locks when it has terminated.
sealed class Thread {
    Thread(ThreadStart start);
    void Start();
    bool Join(int msec);
    static void Sleep(int msec);
    void Abort();
    static void ResetAbort();
    void Interrupt();
    void Suspend();
    void Resume();
    ...
}

sealed class Monitor {
    private Monitor() { }
    ... static void Enter(object obj);
    static void Exit(object obj);
    static bool Wait(object obj, int msec);
    static void Pulse(object obj);
    static void PulseAll(object obj);
    ... static void ResetAbort();
    void Interrupt();
    void Suspend();
    void Resume();
    }

3 An ASM Model for Threads on Uniprocessors

Whenever in C# an object is created on the heap, it gets two additional overhead fields associated with it. The first field is a pointer to the object’s method table. This pointer makes it possible to obtain the run-time type (exact type) of the object. The second field contains an index of a SyncBlock. SyncBlocks are associated with an object on the fly when the object is used as a monitor. A SyncBlock structure contains information that is used for thread synchronization (cf. [14]).

3.1 The Vocabulary for Threads and Monitors

We abstract from implementation details and assume that the dynamic function runTimeType: Ref → Type returns for every object reference its run-time type. The set of threads can then be defined as follows:

$$Thread = \{ ref \in Ref \mid \text{runTimeType}(ref) = \text{Thread}\}$$

The subuniverse Monitor ⊆ Ref is equipped with a dynamic function lockOwner which returns the thread that currently owns the lock of the monitor, a lockCount which counts how many times a thread has to exit the monitor before the lock is released, a readyQueue (also known as lock queue) which holds the ordered queue of blocked threads that are ready to acquire the lock, and a waitQueue which holds the ordered queue of threads that are waiting on the monitor.

$$\text{lockOwner}: \text{Monitor} \rightarrow \text{Thread} \cup \{\text{None}\}$$
$$\text{lockCount}: \text{Monitor} \times \text{Thread} \rightarrow \mathbb{N}$$
$$\text{readyQueue}: \text{Monitor} \rightarrow \text{List(\text{Thread})}$$
$$\text{waitQueue}: \text{Monitor} \rightarrow \text{List(\text{Thread})}$$

When an object ref is used as a Monitor the functions are initialized as follows:

$$\text{lockOwner}(\text{ref}) := \text{None} \hspace{1cm} \text{readyQueue}(\text{ref}) := []$$
$$\text{lockCount}(\text{ref}, \text{thread}) := \text{Undef} \hspace{1cm} \text{waitQueue}(\text{ref}) := []$$

The possible execution states of a thread (explained in detail below) are:

$$\text{ExecState} ::= \text{Unstarted} \mid \text{Active} \mid \text{Suspended} \mid \text{Sleeping} \mid \text{Joined}$$
$$\mid \text{Syncing} \mid \text{Waiting} \mid \text{Pulsed} \mid \text{Dead}$$
The function \textit{execState} returns the unique execution state of a thread.

\textit{execState}: \text{Thread} \rightarrow \text{ExecState}

Every thread has several attributes. The \textit{joinSet} comprises the threads that are joined to the current thread and are waiting for its termination. The \textit{wakeupTime} stores the time when the current thread expires. The \textit{monObj} is the monitor the current thread is waiting for or wants to acquire. The \textit{joinedThread} is another thread on which the current thread is joined. Moreover, several flags indicate whether an abort has been requested or initiated, whether an interrupt has been requested, or whether a suspend has been requested.

\textit{joinSet}: \text{Thread} \rightarrow \text{Powerset(Thread)} \quad \textit{abortRequested}: \text{Thread} \rightarrow \text{Bool}
\textit{wakeupTime}: \text{Thread} \rightarrow \mathbb{N} \cup \{\infty\} \quad \textit{abortInitiated}: \text{Thread} \rightarrow \text{Bool}
\textit{monObj}: \text{Thread} \rightarrow \text{Monitor} \quad \textit{interruptRequested}: \text{Thread} \rightarrow \text{Bool}
\textit{joinedThread}: \text{Thread} \rightarrow \text{Thread} \quad \textit{suspendRequested}: \text{Thread} \rightarrow \text{Bool}

When an object \textit{ref} of type \text{Thread} is created, the dynamic functions are initialized as follows:

\text{joinSet}(\text{ref}) := \emptyset \quad \text{abortRequested}(\text{ref}) := \text{False}
\text{wakeupTime}(\text{ref}) := \text{Undef} \quad \text{abortInitiated}(\text{ref}) := \text{False}
\text{monObj}(\text{ref}) := \text{Undef} \quad \text{interruptRequested}(\text{ref}) := \text{False}
\text{joinedThread}(\text{ref}) := \text{Undef} \quad \text{suspendRequested}(\text{ref}) := \text{False}
\text{execState}(\text{ref}) := \text{Unstarted}

The local state of a thread comprises a frame stack of activation records, the currently executed method, the current position in the method body (program counter), the local environment and the already computed values of expressions (operand stack).

\textit{frames}: \text{Thread} \rightarrow \text{List(Frame)} \quad \textit{locals}: \text{Thread} \rightarrow (\text{Loc} \rightarrow \text{Adr})
\textit{meth}: \text{Thread} \rightarrow \text{Meth} \quad \textit{values}: \text{Thread} \rightarrow (\text{Pos} \rightarrow \text{Result})
\textit{pos}: \text{Thread} \rightarrow \text{Pos}

The current thread is denoted by \texttt{self} in the ASM rules below.

Fig. 3 shows a classification of the execution states of a thread and relates them to the items of the \textit{ThreadState} enumeration in Fig. 1. A thread is \text{Running} if it is not \text{Unstarted} and not already \text{Dead}. A thread is considered to be \text{Passive} (or \text{WaitSleepJoin}) if it is \text{Running} but neither \text{Active} nor \text{Suspended}.

\text{Running}(\text{thread}) \iff \text{execState}(\text{thread}) \notin \{\text{Unstarted}, \text{Dead}\}
\text{Passive}(\text{thread}) \iff \text{WaitSleepJoin}(\text{thread}) \iff 
\quad \text{execState}(\text{thread}) \in \{\text{Syncing}, \text{Waiting}, \text{Pulsed}, \text{Sleeping}, \text{Joined}\}

The items \text{Stopped} and \text{Aborted} of the \textit{ThreadState} enumeration can be obtained as follows:

\text{Stopped}(\text{thread}) \iff \text{execState}(\text{thread}) = \text{Dead} \land \neg \text{abortRequested}(\text{thread})
\text{Aborted}(\text{thread}) \iff \text{execState}(\text{thread}) = \text{Dead} \land \text{abortRequested}(\text{thread})

The reason for this separation is not known to us.
3.2 An Overview of the Model

Fig. 4 and 5 contain diagrams for the execution states of a thread. Methods that are invoked by another thread on the current thread are displayed in grey boxes, whereas methods invoked by the current thread itself are put into white boxes. If there is no outgoing arrow for a thread method from an execution state, then this can mean either that such an invocation is not possible, e.g. since a static method can only be invoked by an active thread, or that the invocation is not allowed and throws a ThreadStateException.

The main rule of the ASM model for uniprocessors in Sect. 3.5 below is the rule ExecSEQUENTIALCSharp. It uses the rule ExecCSharp of the ASM model in [2] which executes one computation step of a single-threaded C# program. The rule ExecCSharp has to be parameterized by the current thread in order to extend the model of [2] to multiple threads. The argument thread of ExecCSharp becomes then the value of ‘self’ in the rules of [2]. The component ExecCSharp

\[ T \equiv \text{ExecCSharp}^{\text{Thread}} \]

3.3 The Methods of the Thread Class

When a thread is created by invoking the constructor of the Thread class, its execution state is Unstarted. The new thread is later started by invoking the Thread.Start method. A thread can only be started once, otherwise a ThreadStateException exception is thrown. If there has already been an abort requested for the thread, its execution state is immediately changed to Dead. Otherwise, the execution state of the thread is updated to Active and its local state is initiated. The new thread now runs concurrently with the thread that invoked the Thread.Start method. The YieldUp(Norm) means that the Thread.Start method returns without blocking.
**ThreadStart(thread)**  ≡  
  if execState(thread) ≠ Unstarted then  FAILUP(ThreadStateException)
  else
    if abortRequested(thread) then  execState(thread) := Dead
    else { THREADINIT(thread),  execState(thread) := Active }
    YIELDUP(Norm)

When a thread is created, a delegate of type ThreadStart has to be provided to the constructor of the Thread class. This delegate is later invoked, when the thread is started. Technically, this means the new thread executes the Invoke method of the ThreadStart delegate. If the invocation list of that delegate consists of a single method, then this method is executed. Otherwise, the methods of the invocation list are executed sequentially.

**ThreadInit(thread)**  ≡  
  let d = getField(thread, Thread::delegate)
  let m = ThreadStart::Invoke() in
    frames(thread) := []
    meth(thread) := m
    pos(thread) := body(m)
    values(thread) := ∅
    INITLOCALS(thread, m, [d])

The INITLOCALS macro initializes the local environment of the method, for example it assigns the delegate d to the this parameter of Invoke. 

The Thread.Join method puts the current thread into the join set of another thread and changes the execution state of the current thread from Active to Joined. Like every thread method that takes a timeout argument it checks first whether the argument is in the correct range. If an interrupt has been requested,
Fig. 5. Methods invoked by the current thread.
then `Thread.Join` throws a `ThreadInterruptedException` instead of joining (Mono 0.26 ignores the interrupt request [12]).

```
ThreadJoin(thread, msec) ≡
  if msec < −1 then FailUp(ArgumentOutOfRangeException)
  elseif execState(thread) = Unstarted then
    FailUp(ThreadStateException)
  elseif execState(thread) = Dead then YieldUp(True)
  elseif interruptRequested(self) then ThrowInterruptedException
  else
    SetWakeupTime(msec)
    joinSet(thread) := joinSet(thread) ∪ {self}
    joinedThread(self) := thread
    execState(self) := Joined
```

The thread will become active again, when the other thread has terminated or `msec` milliseconds have passed. An argument of −1 milliseconds means an infinite amount of time.

```
SetWakeupTime(msec) ≡
  if msec = −1 then wakeupTime(self) := ∞
  else wakeupTime(self) := currentTime + msec
```

When an `ThreadInterruptedException` is thrown, the interrupt request of the current thread is cleared.

```
ThrowInterruptedException ≡
  FailUp(ThreadInterruptedException)
  interruptRequested(self) := False
```

When the `Thread.Join` method returns, it indicates with a boolean result, whether the other thread is dead. If the other thread is not dead, then it follows from the definition of the predicate `Expired` and the `WAKEUP` rule in Sect. 3.5 that the amount of time has expired.

```
ThreadJoinReturn ≡
  if execState(joinedThread(self)) = Dead then YieldUp(True)
  else YieldUp(False)
```

The `Thread.Sleep` method puts the current thread to sleep for the specified amount of milliseconds. The execution state of the current thread is changed from `Active` to `Sleeping`. If an interrupt has been requested, the current thread throws a `ThreadInterruptedException` instead of going to sleep.

```
ThreadSleep(msec) ≡
  if msec < −1 then FailUp(ArgumentOutOfRangeException)
  elseif interruptRequested(self) then ThrowInterruptedException
  else
    execState(self) := Sleeping
    SetWakeupTime(msec)
    YieldUp(Norm)
```
In order to abort another thread with the `Thread.Abort` method, the current thread needs the appropriate security permission. If the other thread is suspended, a `ThreadStateException` is thrown, although the documentation [11] says that in that case the other thread is resumed by the system. Otherwise `Thread.Abort` sets an abort request for the other thread. The effect of this request is that a `ThreadAbortException` is thrown asynchronously in the other thread (see Sect. 3.5).

\[
\text{ThreadAbort}(thread) \equiv \\
\text{if } \neg \text{SecurityPermission}(\text{self}, \text{ControlThread}) \text{ then} \\
\hspace{1em} \text{FailUp(SecurityException)} \\
\text{elseif execState(thread) = Suspended then} \\
\hspace{1em} \text{FailUp(ThreadStateException)} \\
\text{else} \\
\hspace{2em} \text{if } \neg \text{abortRequested(thread)} \land \neg \text{abortInitiated(thread)} \text{ then} \\
\hspace{3em} \text{abortRequested(thread)} := \text{True} \\
\hspace{3em} \text{YieldUp(Norm)}
\]

The static method `Thread.ResetAbort` can be invoked by the current thread to cancel the automatic re-throwing of a `ThreadAbortException` at the end of catch blocks. It clears the flag that indicates that the abort has been initiated. Only threads that have the appropriate security permission can cancel an abort. The documentation [11] says that the method throws a `ThreadStateException` if the method was not invoked on the current thread. This can never happen, since it is a static method.

\[
\text{ThreadResetAbort} \equiv \\
\text{if } \neg \text{SecurityPermission}(\text{self}, \text{ControlThread}) \text{ then} \\
\hspace{1em} \text{FailUp(SecurityException)} \\
\text{elseif } \neg \text{abortInitiated(self)} \text{ then} \hspace{1em} \text{FailUp(ThreadStateException)} \\
\text{else } \{ \text{abortInitiated(self)} := \text{False}, \text{YieldUp(Norm)} \}
\]

The `Thread.Interrupt` method sets an interrupt request for another thread. The effect of the request is that a `ThreadInterruptedException` is injected into the other thread, if it is in a passive state (see Sect. 3.5). Otherwise, the exception is thrown by the other thread, when it changes its execution state from running into a passive state. If the other thread stays active forever, the interrupt request is ignored.

\[
\text{ThreadInterrupt}(thread) \equiv \\
\text{if } \neg \text{SecurityPermission}(\text{self}, \text{ControlThread}) \text{ then} \\
\hspace{1em} \text{FailUp(SecurityException)} \\
\text{else} \\
\hspace{2em} \text{if } \neg \text{interruptRequested(thread)} \text{ then} \\
\hspace{3em} \text{interruptRequested(thread)} := \text{True} \\
\hspace{3em} \text{YieldUp(Norm)}
\]

The `Thread.Suspend` method sets a suspend request for another thread, if it is running. The request will asynchronously be processed by the run-time system (see Sect. 3.5). A suspend request on an already suspended thread has no effect.
The lock is free and the readyQueue of the monitor is empty, the thread immediately gets the lock. Otherwise, the thread changes its state from Active to Syncing and the thread is added to the readyQueue of the monitor. In case of a pending interrupt, a ThreadInterruptedException is thrown.

\textsc{MonitorEnter}(\textit{mon}) \equiv \\
\begin{aligned}
\text{if } \text{null} \text{ then } & \text{FAILUP}(\text{ArgumentNullException)} \\
\text{elseif } \text{lockOwner}(\textit{mon}) = & \text{self} \text{ then} \\
\text{lockCount}(\textit{mon}, \text{self}) & := \text{lockCount}(\textit{mon}, \text{self}) + 1 \\
\text{yieldUP}(\text{Norm}) \\
\text{elseif } \text{lockOwner}(\textit{mon}) = \text{None} \wedge & \text{Empty} (\text{readyQueue}(\textit{mon})) \text{ then} \\
\text{Lock}(\text{self}, \textit{mon})
\end{aligned}

A thread can be resumed by invoking \texttt{Thread.Resume} only if it is suspended or a suspend request is pending. If the thread is suspended, then its execution state is changed back to active such that it can be scheduled for execution by the run-time system again. If a suspend has been requested, the request is cleared.

\textsc{ThreadResume}(\textit{thread}) \equiv \\
\begin{aligned}
\text{if } \text{null} \text{ then } & \text{FAILUP}(\text{ArgumentNullException)} \\
\text{elseif } \text{execState}(\textit{thread}) \neq & \text{Suspended} \wedge \neg \text{suspendRequested}(\textit{thread}) \\
& \text{then } \text{suspendRequested}(\textit{thread}) := \text{True} \\
& \text{yieldUP}(\text{Norm}) \\
\text{elseif } \text{execState}(\textit{thread}) = & \text{Suspended} \text{ then } \text{execState}(\textit{thread}) := \text{Active} \\
\text{if } \text{suspendRequested}(\textit{thread}) & \text{ then } \text{suspendRequested}(\textit{thread}) := \text{False} \\
& \text{yieldUP}(\text{Norm})
\end{aligned}

If a \texttt{ThreadSuspend}(\textit{t}) is executed in parallel with a \texttt{ThreadResume}(\textit{t}) on a thread \(t\) that already has a suspend request, the resume has priority over the suspend\(^1\).

### 3.4 The Methods of the Monitor Class

The methods of the \texttt{Monitor} class are static and are used to acquire and release locks of monitors. If they are invoked with the \texttt{null} reference, an exception is thrown. The \texttt{Monitor.Enter} method is used to acquire the lock of a monitor. If the current thread already owns the lock, the \texttt{lockCount} is increased. If the lock is free and the \texttt{readyQueue} of the monitor is empty, the thread immediately gets the lock. Otherwise, the thread changes its state from \texttt{Active} to \texttt{Syncing} and the thread is added to the \texttt{readyQueue} of the monitor. In case of a pending interrupt, a \texttt{ThreadInterruptedException} is thrown.

\textsc{MonitorEnter}(\textit{mon}) \equiv \\
\begin{aligned}
\text{if } \text{null} \text{ then } & \text{FAILUP}(\text{ArgumentNullException)} \\
\text{elseif } \text{lockOwner}(\textit{mon}) = & \text{self} \text{ then} \\
\text{lockCount}(\textit{mon}, \text{self}) & := \text{lockCount}(\textit{mon}, \text{self}) + 1 \\
\text{yieldUP}(\text{Norm}) \\
\text{elseif } \text{lockOwner}(\textit{mon}) = \text{None} \wedge & \text{Empty} (\text{readyQueue}(\textit{mon})) \text{ then} \\
\text{Lock}(\text{self}, \textit{mon})
\end{aligned}

\(^1\) One could as well forbid the parallel execution by a run constraint (see Sect. 4.1).
lockCount(mon, self) := 1
YIELDUp(Norm)
elseif interruptRequested(self) then THROWINTERRUPTEDEXCEPTION
else
  readyQueue(mon) := readyQueue(mon) · [self]
  monObj(self) := mon
  execState(self) := Syncing
  YIELDUp(Norm)

To lock a monitor means to update the lockOwner of the monitor.

LOCK(thread, mon) ≡ lockOwner(mon) := thread

The Monitor.Exit method decrements the lock count by one. If the lock count becomes 0, the lock is released.

MONITOREXIT(mon) ≡
  if mon = Null then FAILUp(ArgumentNullException)
  elseif lockOwner(mon) ≠ self then
    FAILUp(SynchronizationLockException)
  else
    if lockCount(mon, self) = 1 then UNLOCK(self, mon)
    lockCount(mon, self) := lockCount(mon, self) − 1
    YIELDUp(Norm)

To release a lock means to update the lockOwner of the monitor to None.

UNLOCK(thread, mon) ≡ lockOwner(mon) := None

The documentation [11] says that a thread can only exit a monitor if it owns the lock. The following code, however, runs in the .NET framework 1.1 as well as in Rotor [16] without throwing a SynchronizationLockException.

Object o = new object();
Monitor.Enter(o);
Monitor.Exit(o);
Monitor.Exit(o); // Bug. Thread does not own lock.

The Monitor.Wait method appends the current thread to the waitQueue of the monitor and temporarily releases the lock of the monitor. The execution state of the current thread is changed from Active to Waiting.

MONITORWAIT(mon, msec) ≡
  if mon = Null then FAILUp(ArgumentNullException)
  elseif msec < −1 then FAILUp(ArgumentOutOfRangeException)
  elseif lockOwner(mon) ≠ self then
    FAILUp(SynchronizationLockException)
  elseif interruptRequested(self) then THROWINTERRUPTEDEXCEPTION
  else
    SETWAKEUPTIME(msec)
    waitQueue(mon) := waitQueue(mon) · [self]
    UNLOCK(self, mon)
    monObj(self) := mon
    execState(self) := Waiting
The thread remains in the `waitQueue` of the monitor until the monitor is pulsed or `msec` milliseconds have passed. At the return, the method indicates with a boolean result whether the time has expired.

\[
\text{MonitorWaitReturn} \equiv \\
\begin{array}{ll}
\text{if } \text{wakeupTime}(\text{self}) < \text{currentTime} \text{ then } \text{YieldUp}(\text{True}) \\
\text{else } \text{YieldUp}(\text{False})
\end{array}
\]

The `Monitor.Pulse` method moves the first element of the `waitQueue` of the monitor to the `readyQueue`. If the `waitQueue` is empty, the method just returns. Note, that we allow also that a thread with an abort or an interrupt request can be pulsed (this point is discussed also in [10]).

\[
\text{MonitorPulse}(\text{mon}) \equiv \\
\begin{array}{ll}
\text{if } \text{mon} = \text{Null} \text{ then } \text{FailUp}(\text{ArgumentNullException}) \\
\text{elseif lockOwner}(\text{mon}) \neq \text{self} \text{ then } \\
\text{FailUp}(\text{SynchronizationLockException}) \\
\text{else } \\
\text{if } \neg \text{Empty}(\text{waitQueue}(\text{mon})) \text{ then } \\
\text{MoveToReadyQueue}(\text{first}(\text{waitQueue}(\text{mon})), \text{mon}) \\
\text{YieldUp}(\text{Norm})
\end{array}
\]

When a thread is moved from the `waitQueue` to the `readyQueue`, its execution state is changed from Waiting to Pulsed.

\[
\text{MoveToReadyQueue}(\text{thread}, \text{mon}) \equiv \\
\begin{array}{ll}
\text{readyQueue}(\text{mon}) := \text{readyQueue}(\text{mon}) \cdot [\text{thread}] \\
\text{waitQueue}(\text{mon}) := \text{delete}(\text{thread}, \text{waitQueue}(\text{mon})) \\
\text{execState}(\text{thread}) := \text{Pulsed}
\end{array}
\]

The `Monitor.PulseAll` methods moves all waiting threads into the `readyQueue` of the monitor.

\[
\text{MonitorPulseAll}(\text{mon}) \equiv \\
\begin{array}{ll}
\text{if } \text{mon} = \text{Null} \text{ then } \text{FailUp}(\text{ArgumentNullException}) \\
\text{elseif lockOwner}(\text{mon}) \neq \text{self} \text{ then } \\
\text{FailUp}(\text{SynchronizationLockException}) \\
\text{else } \\
\text{forall } \text{thread} \in \text{waitQueue}(\text{mon}) \text{ do } \text{execState}(\text{thread}) := \text{Pulsed} \\
\text{readyQueue}(\text{mon}) := \text{readyQueue}(\text{mon}) \cdot \text{waitQueue}(\text{mon}) \\
\text{waitQueue}(\text{mon}) := [] \\
\text{YieldUp}(\text{Norm})
\end{array}
\]

In Java, the wait and the ready queues are not FIFO queues but unordered sets. The `Object.notify` method of Java chooses an arbitrary element from the wait set of an object and it is not guaranteed that every thread in the wait set is ever chosen. The proposal for the new Java memory model [10] even allows so-called spurious wake-ups. This means that the system is allowed to remove a thread from the wait set of an object without any reason. Note, that the POSIX thread function `pthread_cond_signal()` is also allowed to wake up more than one thread.
3.5 Scheduling of Threads, Timing, Locking and Asynchronous Exceptions

Although priorities can be assigned to threads in C#, it is not guaranteed that they are honored by the scheduling algorithm. The main ASM rule for sequential C# therefore chooses repeatedly one of the possible threads and executes one (small) step in the computation of the thread. In this way the computation steps of the currently running threads are interleaved.

**ExecSequentialCsharp** ≡
choose thread ∈ Thread do ExecStep(thread)

The next computation step of a thread depends on its current execution state.

**ExecStep(thread)** ≡
if execState(thread) = Active then ExecActive(thread)
elseif Passive(thread) ∧ HasRequest(thread) then
    AbortOrInterruptPassive(thread)
elseif Expired(thread) then Wakeup(thread)
elseif CanAcquireLock(thread) then AcquireLock(thread)

A passive thread has a request, if it has an abort or an interrupt request.

HasRequest(t) ⇔
(abortRequested(t) ∧ ¬CatchFinallyCode(t)) ∨ interruptRequested(t)

A thread is expired, if its wakeup time has been passed. The system time in milliseconds is given by the monitored function currentTime.

Expired(thread) ⇔ execState(thread) ∈ \{Waiting, Sleeping, Joined\} ∧
    wakeupTime(thread) ≤ currentTime

A thread can acquire the lock, if it can acquire the lock of its current monitor object monObj that was set in MONITERENTER or MONITERWAIT.

CanAcquireLock(thread) ⇔ CanAcquireLock(thread, monObj(thread))

The lock of a monitor can be acquired, if the lock is free and the thread is the first thread in the readyQueue of the monitor.

CanAcquireLock(thread, mon) ⇔
    execState(thread) ∈ \{Syncing, Pulsed\} ∧ lockOwner(mon) = None ∧
    thread = first(readyQueue(mon))

When the execution state of the chosen thread is Active, the next step in the computation of the thread is executed by the rule ExecCsharp(thread) already mentioned in Sect. 3.2. In case of a pending abort the system waits until the thread has left any finally block or catch clause before it aborts the threads. In case of a pending suspend, the system waits until the thread reaches a so-called safe point before suspending the thread. Safe points are points that are also safe for garbage collection.

**ExecActive(thread)** ≡
if abortRequested(thread) ∧ ¬CatchFinallyCode(thread) then
    AbortActive(thread)
\textbf{elseif} suspendRequested(thread) \&\& SafePoint(thread) \textbf{then}
\hspace{1em}SUSPEND(thread)
\textbf{else} EXECCSHARP(thread)

A thread aborts by throwing a \texttt{ThreadAbortException}. The fact that the thread is responding to the abort request is recorded in the \texttt{abortInitiated} flag.

\texttt{ABORTACTIVE(thread) \equiv}
\hspace{1em}FAIL(thread, ThreadAbortException)
\hspace{1em}CLEARABORTREQUEST(thread)
\hspace{1em}abortInitiated(thread) := True

When the abort request is cleared, any pending interrupt request is also cleared.

\texttt{CLEARABORTREQUEST(thread) \equiv}
\hspace{1em}abortRequested(thread) := False
\hspace{1em}if interruptRequested(thread) \textbf{then}
\hspace{2em}interruptRequested(thread) := False

Like any other exception, a \texttt{ThreadAbortException} is propagated upwards in the frame stack of the thread. If it crosses a try block with catch clauses and a possible finally block, the catch clauses are searched for a matching handler. If there exists one, the corresponding catch block is executed. The finally block is executed afterwards. At the end of the catch block, however, the \texttt{ThreadAbortException} is re-thrown by the system. More precisely, if an abort has been initiated and a catch block terminates but not with an exception, then a new \texttt{ThreadAbortException} is thrown at the end of the catch block. If the catch block terminates abruptly with an exception, that exception is propagated upwards. Hence, the rules for exception handling of C\# of [2] have to be refined. If the current position of the thread (indicated by the black triangle) is at the end of a catch block with result \texttt{res} and \texttt{res} is not an exception, a \texttt{ThreadAbortException} is thrown. Hence a \texttt{ThreadAbortException} cannot be swallowed unless the \texttt{Thread.ResetAbort} method is called (see Sect. 3.3).

\texttt{EXECCHSHRPSTM}_{T} \equiv \textbf{match} context(pos)
\hspace{1em}\texttt{try} Exc(ref) \ldots catch(...) \triangleright res \ldots \rightarrow
\hspace{2em}excStack := pop(excStack)
\hspace{2em}if abortInitiated(self) \&\& res \notin Exc \textbf{then}
\hspace{3em}FAILUP(ThreadAbortException)
\hspace{1em}\textbf{else} YIELDUP(res)

When a suspend request is pending and the thread has reached a safe point, the system changes its execution state from \textit{Active} to \textit{Suspected} and clears the request.

\texttt{SUSPEND(thread) \equiv}
\hspace{1em}execState(thread) := Suspected
\hspace{1em}suspendRequested(thread) := False

If a passive thread has an abort or an interrupt request, an exception is injected into the thread. If both, an abort request and an interrupt request are pending, the abort request has priority (unless the thread executes catch or finally code).
AbortOrInterruptPassive(thread) \equiv \\
\quad \text{if } abortRequested(thread) \land \neg \text{CatchFinallyCode(thread)} \text{ then} \\
\quad \text{InjectException(thread, ThreadAbortException)} \\
\quad \text{ClearAbortRequest(thread)} \\
\quad \text{abortInitiated(thread) := True} \\
\quad \text{elseif } interruptRequested(thread) \text{ then} \\
\quad \text{InjectException(thread, ThreadInterruptedException)} \\
\quad \text{interruptRequested(thread) := False}

Injecting a ThreadAbortException or a ThreadInterruptedException into a thread means to create a new exception object and to force the thread to throw the exception (using FAIL) and to wakeup the thread.

InjectException(thread, exception) \equiv \\
\quad \text{FAIL(thread, exception)} \\
\quad \text{Wakeup(thread)}

If an exception is injected into a passive thread or if a sleeping, joined or waiting thread has expired, the system has to wakeup the thread. If the thread is Sleeping, its execution state is changed to Active. If the thread is Joined, it is removed from the joinSet and returns from the Thread.Join method (see Sect. 3.3). If the thread is Syncing, its execution state is changed to Active and the thread is removed from the readyQueue of the monitor. If the thread is Waiting, it is moved from the waitQueue to the readyQueue and has to re-acquire the lock in the Pulsed state, since in this case the thread is still in a critical section of code and possible exception handlers and finally blocks should only be executed under the exclusive control of the monitor. If the thread is Pulsed, its execution state is not updated, since the thread has to re-acquire the lock before executing any further code.

Wakeup(thread) \equiv \\
\quad \text{if } execState(thread) \in \{ \text{Sleeping, Joined, Syncing} \} \text{ then} \\
\quad \quad \text{execState(thread) := Active} \\
\quad \quad \text{if } execState(thread) = \text{Joined} \text{ then let } t = \text{joinedThread(thread) in} \\
\quad \quad \quad \text{joinSet(t) := joinSet(t) \setminus \{ thread \}} \\
\quad \quad \text{if } execState(thread) = \text{Syncing} \text{ then let } mon = \text{monObj(thread) in} \\
\quad \quad \quad \text{readyQueue(mon) := delete(thread, readyQueue(mon))} \\
\quad \quad \text{if } execState(thread) = \text{Waiting} \text{ then} \\
\quad \quad \quad \text{MoveToReadyQueue(thread, monObj(thread))}

If a thread can acquire the lock, it becomes the owner of the lock. Its execution state is changed from Pulsed or Syncing to Active. If it acquires the lock for the first time, the lockCount is initialized to 1. Otherwise, when the thread has temporarily released the lock by invoking the Monitor.Wait method, the old lock count is still valid.
ACQUIRELOCK(thread) ≡
  let mon = monObj(thread) in
  Lock(thread, mon)
  if execState(thread) = Syncing then lockCount(mon, thread) := 1
  readyQueue(mon) := tail(readyQueue(mon))
  execState(thread) := Active

A thread terminates when the frame stack of the thread is empty again and
the Invoke method of the delegate of the thread terminates. The method can
terminate normally or abruptly with an exception. In any case, the execution
state of the thread is updated from Active to Dead and the threads that are
joined are notified by changing their states from Joined to Active such that
they can return from the Thread.Join method. If the thread terminates with
an exception, the exception may or may not be reported as unhandled exception
to the console.

ExecCsharpStm,T ≡ match context(pos)
  res → if pos = body(meth) ∧ Empty(frames) then
    forall thread ∈ joinSet(self) do execState(thread) := Active
    joinSet(self) := Ø
    execState(self) := Dead
  if exception(res) then REPORTUNHANDLEDEXC(res)

When a thread is Dead, it remains in this execution state and cannot be re-
activated (see also Fig. 4).

3.6 Invariants of the Model

The following invariants are satisfied (where t ∈ Thread and mon ∈ Ref):

(thread) x ∈ Thread ↔ execState(x) ≠ Undef
(join1) execState(t) ∈ {Unstarted, Dead} ⇒ joinSet(t) = Ø
(join2) execState(t) = Joined ⇒ t ∈ joinSet(joinedThread(t))
(join3) x ∈ joinSet(t) ⇒ x ∈ Thread ∧ execState(x) = Joined
(lock1) lockCount(mon, t) ∈ ℕ ∪ {Undef}
(lock2) lockCount(mon, x) ≠ Undef ⇒ x ∈ Thread
(lock3) execState(t) = Unstarted ⇒ lockCount(mon, t) = Undef
(lock4) lockOwner(mon) ≠ None ⇒
  lockOwner(mon) ∈ Thread ∧ lockCount(mon, lockOwner(mon)) ≥ 1
(lock5) lockCount(mon, t) ≥ 1 ⇒
  lockOwner(mon) = t ∨ t ∈ readyQueue(mon) · waitQueue(mon)
(wait1) execState(t) = Waiting ⇒ lockOwner(monObj(t)) ≠ t ∧
  t ∈ waitQueue(monObj(t)) ∧ lockCount(monObj(t), t) ≥ 1
(wait2) x ∈ waitQueue(mon) ⇒ x ∈ Thread ∧ execState(x) = Waiting
(sync) execState(t) = Syncing ⇒
  t ∈ readyQueue(monObj(t)) ∧ lockCount(monObj(t), t) ∈ {Undef, 0}
(pulsed) $\text{execState}(t) = \text{Pulsed} \implies \text{lockOwner}(\text{monObj}(t)) \neq t \land t \in \text{readyQueue}(\text{monObj}(t)) \land \text{lockCount}(\text{monObj}(t), t) \geq 1$

(ready) $x \in \text{readyQueue}(\text{mon}) \implies x \in \text{Thread} \land \text{execState}(x) \in \{\text{Syncing}, \text{Pulsed}\}$

(abort) $\text{abortInitiated}(t) \implies \neg \text{abortRequested}(t)$

(suspend1) $\text{execState}(t) \in \{\text{Unstarted}, \text{Dead}\} \implies \neg \text{suspendRequested}(t)$

(suspend2) $\text{execState}(t) = \text{Suspended} \implies \neg \text{suspendRequested}(t)$

(time) $\text{execState}(t) \in \{\text{Waiting}, \text{Sleeping}, \text{Joined}\} \implies \text{wakeupTime}(t) \geq 0$

The most important invariant is (lock5) which says that among the threads that have entered a given monitor there exists at most one thread that is not in the ready queue or the wait queue of the monitor. Hence, the locks of monitors are exclusive and can be used to protect critical code sections.

### 4 A Parallel Model for C# Threads

On a multiprocessor system different threads can execute code concurrently on different processors. We model this case using a special kind of distributed ASMs that are executable in tools like AsmL [6]. The main rule for the parallel thread model chooses repeatedly an arbitrary set of possible threads and executes in parallel the next computation step for each of the chosen threads.

$$\text{ExecParallelCsharp} \equiv$$

$$\text{choose } T \subseteq \text{Thread} \text{ do }$$

$$\text{forall } \text{thread } \in T \text{ do } \text{ExecStep(thread)}$$

The ExecStep rule is the same as in the sequential model. However, since the computation steps are executed in parallel, conflicts can occur if the same location is updated by different threads to different values. Updates of the local state of a thread (frame stack, program counter, local environment, operand stack) are not critical, since the local state is parameterized by the thread (see Sect. 3.1). Updates to the shared memory are discussed in Sect. 5 below. An analysis of the transition rules shows that the following conflicts have to be avoided by imposing run constraints on ExecParallelCsharp.

### 4.1 Run Constraints for the Parallel Thread Model

1. The rule $\text{ACQUIRELOCK}(\text{thread})$ is not allowed to run in parallel with the rule $\text{MONITORENTER}(\text{mon})$, if $\text{monObj}(\text{thread}) = \text{mon}$. Otherwise, there would be a conflict for $\text{lockOwner}(\text{mon})$.

2. It is not allowed that two different threads execute $\text{MONITORENTER}(\text{mon})$ in parallel. Otherwise, conflicts for $\text{lockOwner}(\text{mon})$ and $\text{readyQueue}(\text{mon})$ could occur.

3. The rules $\text{ACQUIRELOCK}(\text{thread})$ and $\text{MOVEToREADYQUEE}(t, \text{mon})$ are not allowed to run in parallel, if $\text{monObj}(\text{thread}) = \text{mon}$. Otherwise, there is a conflict for $\text{readyQueue}(\text{mon})$. 
4. The rule $\text{MonitorPulse}(mon)$ or $\text{MonitorPulseAll}(mon)$ is not allowed to run in parallel with the rule $\text{MoveToReadyQueue}(t, mon)$. Otherwise, there would be a conflict for $\text{readyQueue}(mon)$.

5. $\text{MoveToReadyQueue}(t_1, mon)$ and $\text{MoveToReadyQueue}(t_2, mon)$ are not allowed to run in parallel for $t_1 \neq t_2$. Otherwise, there would be a conflict for $\text{readyQueue}(mon)$.

6. The rules $\text{Suspend}(thread)$ and $\text{ThreadResume}(thread)$ are not allowed to run in parallel. Otherwise, the $\text{ThreadResume}(thread)$ would be ignored by the system.

Note, that $\text{MoveToReadyQueue}(t, mon)$ is used in the rules $\text{Wakeup}(t)$ and $\text{AbortOrInterruptPassive}(t)$ in case that the execution state of $t$ is $\text{Waiting}$.

5 The .NET Memory Model

The .NET memory model is outlined in [5, Partition 1, §11.6.5, §11.6.7]. According to [13,15], it gives the following (weak) guarantees about the ordering of memory reads and writes:

- Reads and writes from the same thread to a location cannot be re-ordered.
- No read can move before a lock acquire (or volatile read).
- No write can move after a lock release (or volatile write).
- Writes cannot cross a $\text{Thread.WriteMemoryBarrier}()$.
- Neither reads nor writes can cross a $\text{Thread.MemoryBarrier}()$.

To application programmers the memory model is often (wrongly) explained in a stronger form. Each thread has its local cache. After acquiring the lock the thread’s cache is invalidated, so that reads afterward are done from the main memory. After releasing the lock the thread’s cache is flushed to main memory. Note that in .NET a read or write of a volatile location affects also the ordering of reads and writes of other locations.

For the ASM specification of the .NET memory model we follow the ASM specification of the Local Consistency Memory Model for Java in [1] and use a universe of events that is divided into disjoint subuniverses as follows:

$$\text{Event} ::= \text{WriteEvent} | \text{LockEvent} | \text{UnlockEvent} | \text{ReadVolatileEvent}$$
$$\text{| BarrierEvent} | \text{WriteBarrierEvent}$$

Events are ordered during a run of a multi-threaded C# program by a dynamic predicate $\prec$. We denote by $\prec^+\text{ the transitive closure and by } \prec^\ast\text{ the reflexive, transitive closure of } \prec$. Each $\text{WriteEvent}$ has two attributes, an address and a value, $\text{adr: WriteEvent} \rightarrow \text{Address val: WriteEvent} \rightarrow \text{Value}$. The latest event of a thread is recorded in $\text{latest: Thread} \rightarrow \text{Event}$.

The memory model is now reduced to the question: “Which write event(s) can be seen by a memory read?” When a thread writes a value to an address, a new $\text{WriteEvent}$ is created.
class Foo {
    private Helper helper;
    public Helper GetHelper() {
        if (helper == null) // quick check
            lock (this)
                if (helper == null) { // double check
                    Helper o = new Helper();
                    Thread.MemoryBarrier();
                    helper = o;
                }
        return helper;
    }
}

Fig. 6. The double checked locking pattern.

\[
\text{WRITE}(adr, val) \equiv \text{let } e = \text{new}(\text{WriteEvent}) \text{ in } \\
\{ \text{adr}(e) := adr, \text{val}(e) := val \} \\
\text{INSERTAFTERLATEST}(\text{self}, e)
\]

The new WriteEvent is inserted in the event order immediately after the latest event of the current thread.

\[
\text{INSERTAFTERLATEST}(\text{thread}, e) \equiv \\
\text{if latest}(\text{thread}) \neq \text{Undef then latest}(\text{thread}) \prec e := \text{True} \\
\text{latest}(\text{thread}) := e
\]

Reading a value from an address means choosing an appropriate WriteEvent for that address and returning the value that has been written to that address.

\[
\text{READ}(adr) \equiv \\
\text{choose } e \in \text{WriteEvent with } \text{adr}(e) = adr \land \neg \text{Overwritten}(\text{self}, e) \text{ do } \\
\text{return } \text{val}(e)
\]

A read cannot see arbitrary write events but only those that are not overwritten with respect to the latest event of the current thread or any memory barrier.

\[
\text{Overwritten}(t, e) \iff \\
\exists w \in \text{WriteEvent}(\text{adr}(w) = \text{adr}(e) \land e \prec^+ w \land \text{Previous}(t, w))
\]
\[
\text{Previous}(t, w) \iff w \prec^* \text{latest}(t) \lor \exists b \in \text{WriteBarrierEvent}(w \prec^* b)
\]

When a monitor is locked a new LockEvent is created and inserted in the event order after the latest UnlockEvent of the monitor as well as after the latest event of the current thread (in this way the write events of the last thread that owned the lock are synchronized with the current thread).

\[
\text{LOCK}(\text{thread}, \text{mon}) \equiv \text{let } e = \text{new}(\text{LockEvent}) \text{ in } \\
\text{if latestUnlock}(\text{mon}) \neq \text{Undef then latestUnlock}(\text{mon}) \prec e := \text{True} \\
\text{INSERTAFTERLATEST}(\text{thread}, e) \\
\text{forall } b \in \text{WriteBarrierEvent do } b \prec e := \text{True} \\
\text{lockOwner}(\text{mon}) := \text{thread}
\]

The function latestUnlock: Monitor \(\rightarrow\) UnlockEvent records the latest unlock event of a monitor. The UnlockEvent created at the monitor exit prevents over-
written write events from being seen by the next thread that acquires the lock of
the monitor.

\[
\text{UNLOCK}(\text{thread}, \text{mon}) \equiv \text{let } e = \text{new(UnlockEvent)} \text{ in} \\
\text{latestUnlock}(\text{mon}) := e \\
\text{INSERTAFTERLATEST}(\text{thread}, e) \\
\text{lockOwner}(\text{mon}) := \text{None}
\]

The function \text{Thread.MemoryBarrier} creates a new \text{BarrierEvent} which is inserted in the event order after the latest event of the current thread. (Barrier events are used in the definition of the \text{Overwritten} predicate above.)

\[
\text{MEMORYBARRIER} \equiv \text{let } e = \text{new(BarrierEvent)} \text{ in} \\
\text{INSERTAFTERLATEST}(\text{self}, e)
\]

The function \text{Thread.WriteMemoryBarrier} creates a \text{WriteBarrierEvent} which are inserted in the event order before any future lock event.

\[
\text{WRITEMEMORYBARRIER} \equiv \text{let } e = \text{new(WriteBarrierEvent)} \text{ in} \\
\text{INSERTAFTERLATEST}(\text{self}, e)
\]

A read of a volatile field creates a new \text{ReadVolatileEvent}. The chosen write event (which was a volatile write) is inserted in the event ordering before the read event.

\[
\text{READVOLATILE}(\text{adr}) \equiv \\
\text{let } r = \text{new(ReadVolatileEvent)} \text{ in} \\
\text{INSERTAFTERLATEST}(\text{self}, r) \\
\forall b \in \text{WriteBarrierEvent} \text{ do } b \prec r := \text{True} \\
\text{choose } e \in \text{WriteEvent} \text{ with } \text{adr}(e) = \text{adr} \land \neg \text{Overwritten}(\text{self}, e) \text{ do} \\
\text{return } \text{val}(e)
\]

A write to a volatile field uses the normal \text{WRITE} rule.

The so-called \textit{double-checked locking pattern} in Fig. 6 uses a memory barrier to prevent another thread from seeing a non-null value of the \text{helper} field while the fields of the \text{Helper} object itself still contain their default null values which are overwritten in the constructor of the \text{Helper} class. (The constructor may be inlined by the JIT compiler.) Instead of using the memory barrier the \text{helper} field could be declared \textit{volatile}.

It is not clear to us, whether the following example is allowed by the Ecma .NET memory model. Consider two threads that concurrently execute the following instructions, where initially \text{p.x} == 0, \text{p.y} == 0:

\[
\begin{align*}
\text{Thread 1} & \quad \text{Thread 2} \\
r1 &= \text{p.x} & r2 &= \text{p.y} \\
\text{p.y} &= 1 & \text{p.x} &= 2
\end{align*}
\]

Is the result \(r1 == 2\) and \(r2 == 1\) possible? According to our specification of the memory model, it is not possible. However, if we allow the compiler to switch the assignments in both threads (under the assumption that \text{p.x} and \text{p.y} are independent variables), the result is plausible. Maybe the result is justified by the paragraph about execution order in [4, §10.10].
class Account {
    private decimal balance = 0.0M;

    public void Deposit() {
        lock (this) {
            try { Monitor.Wait(this); }
            finally { balance += 100.00M; }
        }
    }

    public static void Main() {
        Account a = new Account();
        Thread t = new Thread(new ThreadStart(a.Deposit));
        t.Start();
        Thread.Sleep(100);
        lock (a) {
            Console.WriteLine(a.balance); // Output: 0
            Monitor.Pulse(a);
            t.Interrupt();
            Thread.Sleep(100);
            Console.WriteLine(a.balance); // Output: 100.00 (bug)
        }
    }
}

Fig. 7. A bug in Microsoft’s .NET Framework version 1.1.

6 Conclusion

The ASM method forces the person who writes a specification to think in terms of an abstract implementation. This leads to questions and cases that are usually forgotten in other formal or informal approaches. Fig. 7 contains a bug in Microsoft’s .NET Framework 1.1 [11] which was detected during the construction of our thread model. The bug shows a situation where a thread executes code in a critical sections protected by a monitor without owning the lock of the monitor.

The main function in Fig. 7 creates an account, starts another thread with the Deposit method of the account and sleeps for 100 milliseconds. During the sleep, the deposit thread acquires the lock of the account and waits on the account in order to later deposit 100 dollars when it is pulsed. After the sleep, the main thread locks the account and executes its critical section. At the beginning, the balance is still 0. The main thread pulses the account and moves the deposit thread from the wait queue into the ready queue of the account. Then it interrupts the deposit thread and sleeps again for 100 milliseconds (still holding the lock of the account). When it awakes, the balance has changed to 100.00M. Why?

The change of the balance is only possible if the deposit thread executes the finally block, which is in its critical section, without owning the lock of the account. The same problem occurs if Thread.Interrupt is replaced by Thread.Abort.
The Rotor SSCLI implementation [16] correctly prints 0 at the end of the lock statement in the main function. After that, however, it deadlocks for unknown reasons.

Acknowledgement

We gratefully acknowledge partial support of this work by a Microsoft grant within the ROTOR project during the year 2002–2003.

References

13. V. Morrison. The DOTNET Memory Model. dotnet@discuss.develop.com mailing list, 2002.
Finite Cursor Machines in Database Query Processing

Jan Van den Bussche
Limburgs Universitair Centrum

Abstract. A database system is often concerned with the processing of lists of tuples in a single scan, using constant amount of memory. In classical relational query processing [2], many of the relational algebra operators have simple single-scan implementations on sorted lists. In more recent data stream systems [1], single-scan processing is a must. Data warehousing software tools, such as those by Aruna, support database querying using index structures for text searching.

To improve our understanding of the possibilities and limitations of single-scan, constant-memory processing on lists of tuples, we define and study the abstract model of finite cursor machines. Finite cursor machines are, of course, instantiations of sequential ASMs.

In conjunction with sorting, finite cursor machines can evaluate a wide class of relational algebra expressions; in particular, they can compute all database queries expressible using semijoins rather than full joins.

Challenging problems include delineating the precise computing power of finite cursor machines with sorting, and minimizing the number of sorting operations that are needed. We discuss these problems and present some preliminary results.

This is joint work with Dirk Leinders and Jerzy Tyszkiewicz.

References

Formalizing Liveness-Enriched Sequence Diagrams Using ASMs

Alessandra Cavarra\textsuperscript{1} and Juliana Küster-Filipe\textsuperscript{2,*}

\textsuperscript{1} Computing Laboratory
Oxford University
Wolfson Building, Parks Rd, Oxford OX13QD, UK
Alessandra.Cavarra@comlab.ox.ac.uk

\textsuperscript{2} Laboratory for Foundations of Computer Science, School of Informatics
University of Edinburgh
JCMB, King’s Buildings, Edinburgh EH9 3JZ, UK
jkfilipe@inf.ed.ac.uk

Abstract. In UML 2.0 sequence diagrams have been considerably extended, influenced by Live Sequence Charts (LSCs), a very expressive extension of Message Sequence Charts (MSC) with liveness. Nonetheless, fundamental liveness properties can still not be expressed in the new sequence diagrams. In recent work, we have proposed to solve this by enriching sequence diagrams with a simple but powerful Object Constraint Language (OCL) template for liveness. In this paper, we show how to formalize our liveness-enriched sequence diagrams using Abstract State Machines.

Sequence diagrams still have several open problems. For example, the semantics of some of the newly introduced operators is ambiguous, and it is not clear how they can be used or combined with other operators. We address some of these issues in the paper.

Finally, a further advantage of using ASMs as a semantic model concerns synthesis. It is our ultimate goal to be able to synthesise automatically a state-based object system from our richer sequence diagrams. ASMs are a state-based and operational formalism which therefore eases this task considerably.

1 Introduction and Motivation

When modelling systems using UML, different diagrams and notations are used at different stages of the development process. Unfortunately, there is only a very loose link between the diagrams used at different stages, and no way to move naturally from requirements specifications to detailed design and code.

In analysis, use cases are used at a high-level to capture functional requirements and describe possible scenarios involving the system and its external actors or the environment. However, since use cases are very informal they cannot serve as the basis for a rigorous development process, and a more expressive and formal

* The second author was supported by the EPSRC grant GR/R16891.
language for capturing requirements is needed. Typically, in UML use cases are realised using sequence diagrams, a variant of Message Sequence Charts (MSCs). MSCs have long been adopted as a standard by the International Telecommunication Union (ITU) [12], and are a very popular visual scenario-based language used to specify scenarios as sequences of message interactions between object instances. However, as pointed out by Damm and Harel in [5] MSCs and sequence diagrams in UML1.x have a very weak expressive power. They only describe possible sequences of interactions, and cannot distinguish between possible and mandatory behaviour. Furthermore, there is no way to describe that a certain sequence of interactions should never occur, that is, that a certain sequence of interactions should be forbidden. This is particularly important if we are modelling safety-critical systems.

Basically, MSCs and sequence diagrams in UML1.x can describe (part of) the expected behaviour of a system, but are inadequate to specify the actual behaviour of a system in a scenario-based fashion. The implication of this is ultimately that there is a big gap between the UML notation for specifying inter-object behaviour (sequence diagrams) and intra-object behaviour (state diagrams). It is the latter that corresponds to an implementable model and will lead towards the final software or hardware. Ideally, a more expressive language for capturing requirements as inter-object behaviour would minimise this gap, and make it possible to generate implementable models from a requirements specification.

Live Sequence Charts (LSCs) [5] are a powerful extension of MSCs addressing many of their deficiencies. They can express liveness properties (eventually something must happen) both at the local level (e.g., a message must be received) and at the global level (i.e. referring to a part of the chart, for instance, the interactions described in a subchart must happen). Consequently, LSCs can express possible, mandatory and forbidden behaviour. LSCs have a methodology supported by a tool called Play-Engine [7]. The problem of synthesising a state-based model from an LSC is hard. There are currently no satisfactory algorithms for synthesising state machines from LSCs, but it is believed not to be beyond a possible solution in practice.

In UML 2.0 [11] sequence diagrams have been considerably extended, influenced by LSCs. Nonetheless, fundamental liveness properties can still not be expressed in the new sequence diagrams. In recent work [4], we have proposed to solve this by enriching sequence diagrams with a simple but powerful Object Constraint Language (OCL) template for liveness. This template extends the one given in [2]. As a result liveness-enriched sequence diagrams in UML 2.0 have a comparable expressiveness to LSCs.

In this paper, we show how to formalize our liveness-enriched sequence diagrams using Abstract State Machines (ASMs) [6,1]. There are two advantages in doing so. On the one side, by defining a formal semantics we address some of the semantical ambiguities of the new sequence diagrams in UML 2.0. Sequence diagrams still have several open problems. For example, the semantics of some of the newly introduced operators is ambiguous, and it is not clear how they can
be used or combined with other operators. On the other side, by choosing an operational state-based semantics, we believe that ultimately we are simplifying the problem of synthesising state machines from liveness-enriched sequence diagrams. Such issues are beyond the scope of the present paper, but build the motivation of our work. They are subject for further investigation.

This paper is structured as follows. In the next section, we give an overview of the main features of sequence diagrams in UML 2.0. In section 3, we introduce a basic ASM model for sequence diagrams describing only possible behaviour. A refinement of this model is given in section 4 for sequence diagrams with local liveness and synchronisation. The paper finishes with some conclusions and ideas for further work.

2 Sequence Diagrams in UML 2.0

One of the major changes made to UML 2.0 with respect to its previous versions concerns sequence diagrams that have been extended to include a number of features borrowed by MSCs and, to a limited extent, LSCs. As a consequence, UML’s sequence diagrams are now more expressive. In this section, we give an overview of sequence diagrams in UML 2.0.

Graphically, a sequence diagram has two dimensions: the horizontal dimension represents the instances participating in the scenario; the vertical dimension represents time. Objects have a vertical dashed line called lifeline. The lifeline represents the existence of the instance at a particular time; the order of events along a lifeline is significant denoting the order in which these events will occur.

A message is a communication between two instances that conveys information with the expectation that action will ensue. A message will cause an operation to be invoked, a signal to be raised, or an instance to be created or destroyed. Messages are shown as a horizontal arrows from the lifeline of one instance to the lifeline of another instance. A message specifies not only the kind of communication between instances, but also the sender and receiver event occurrences associated to it.

Sequence diagrams, as in UML 2.0, can contain sub-interactions, called interaction fragments. Interaction fragments can be combined through interaction operators to define an expression\(^1\). The semantics of the resulting combined fragment depends upon the operator and is described informally in the UML 2.0 superstructure specification [11]. Below we give the semantics defined in [11] for some operators that we use in this paper:

- **alt** designates that the combined fragment represents a choice of behaviour. At most one of the operands will execute. The operand that executes must have a guard expression that evaluates to true at this point in the interaction.
- **par** designates that the combined fragment represents a parallel merge between the behaviours of the operands. The event occurrences of the different operands can be interleaved in any way as long as the ordering imposed by each operand as such is preserved.

---

\(^1\) In this case interaction fragments are called interaction operands.
**seq** designates that the combined fragment represents a weak sequencing between the behaviours of the operands, i.e. the ordering of event occurrences within each of the operands are maintained whereas event occurrences on different lifelines from different operands may come in any order, and event occurrences on the same lifeline from different operands are ordered such that an event occurrence of the first operand comes before that of the second operand.

**strict** designates that the combined fragment represents a strict sequencing between the behaviours of the operands. Notationally this means that the vertical coordinate of the contained fragments is significant throughout the whole scope of the combined fragment and not only on one lifeline.

**neg** designates that the combined fragment represents traces that are defined to be invalid. All interaction fragments that are different from negative are considered positive meaning that they describe traces that are valid and should be possible.

**assert** designates that the combined fragment represents an assertion. The sequences of the operand of the assertion are the only valid continuations.

We import in sequence diagrams some concepts introduced in LSCs that we need for our model but are not currently present in sequence diagrams. The lifeline of an instance consists of several points called *locations* corresponding to the occurrence of events. All instances have at least three locations: an initial location, a location corresponding to the beginning of the main chart (the whole diagram for sequence diagrams), and a final location. Locations are also associated with the sending and receiving of messages, the beginning and the end of subcharts (interaction fragments for sequence diagrams), and the evaluation of a condition or an assignment. The locations along a single lifeline are ordered top-down; therefore, a *partial order* is induced among locations determining the order of execution.

Another notion important for LSCs is *temperature*. Every element in an LSC has a temperature which can be either hot or cold. This is used to distinguish between possible (cold) and mandatory (hot) elements and behaviour. An element can be a location, a message and a subchart. Consequently, if a location is hot/cold it must/may be reached; if a message is hot/cold it must/may be received after it has been sent; and if a subchart is hot/cold it describes a collection of interactions that must/may happen.

**Possible Behaviour.** As already mentioned, sequence diagrams can only express the possibility that a certain scenario occurs. In fact, sequence diagrams model behaviour in the form of possible interactions, i.e. communication patterns that *may* occur between a set of instances.

The semantics of an interaction is given as a pair of set of traces, i.e. the set of valid traces and the set of invalid traces. A trace is a sequence of event occurrences denoted \( < e_1, e_2, \ldots, e_n > \). An event occurrence will also include information about the values of all relevant objects at this point in time.
Mandatory Behaviour. UML sequence diagrams, in their current setting, seem to be able to express necessity only to a very limited extent. In particular, it is not clear whether the intention of the new operator called assert is to specify mandatory behaviour. The superstructure specification is ambiguous in the definition of this operator, and it is not obvious from the text whether this operator enforces a sequence of messages to happen, or they are “expected” to happen (see [11], pages 412, 442). However, even if the former case were the intended one, it would still only solve the problem of expressing necessity in sequence diagrams at the interaction level, but not at the local level of a single message or location. Messages and locations in sequence diagrams are implicitly cold. If sent a message may be received, but it does not have to be. Similarly, any location in a sequence diagram may be reached but it does not have to be. This reflects that an instance is not actually forced to progress along its lifeline.

To address the important dichotomy between must and may we have shown in [4] how to achieve necessity, both at the global and the local level, using an extension of OCL for liveness. The idea is that by default a sequence diagram only reflects possible behaviour (except for the assert operator) or possible but forbidden behaviour (given by the neg operator). To impose additionally that a location must be reached or a message must be received, we have to enrich the model with corresponding liveness constraints written in an appropriate OCL template. A more powerful version of the template can also be used to express global liveness. For instance, that after a sequence of interactions has occurred, another sequence of interactions must occur. We omit further details on the OCL constraints in this paper as they are not essential. It suffices to understand that the OCL liveness constraints change the temperature of associated locations from cold to hot.

Synchronisation. Very little is said about when and how instances synchronise in a sequence diagram. Consequently, it is not possible to impose that a certain condition must hold or an object’s attribute must be assigned with a specific value before some specified instances progress in the interaction. Moreover, sequence diagrams allow instances participating in the same interaction fragment to access and exit it independently.

UML 2.0 provides different kinds of conditions in sequence diagrams.

1. An interaction constraint is a boolean expression shown in square brackets covering the lifeline where the first event will occur, positioned above that event inside an interaction operand.

2. A state invariant is a constraint on the state of a lifeline, where by ‘state’ is intended also the values of attributes of the lifeline. The constraint is assumed to be evaluated during runtime immediately prior to the execution of the next event occurrence. If the constraint is true the trace is a valid trace; otherwise, the trace is invalid. Notationally, state invariants are shown as a constraint inside a state symbol or in curly brackets, and are placed on a lifeline.
The problem of synchronisation for interaction constraints can be reduced to the problem of synchronisation for interaction fragments, in that, if instances are synchronised at the beginning of an interaction, then instances are not allowed to progress independently.

As to state invariants, it would be possible to ensure synchronisation among several instances only extending the constraint scope to more than one instance. This is not allowed in UML.

In order to avoid inconsistencies within interaction fragments, we need to enforce synchronisation at their beginning and end lines. This can be done adding corresponding OCL constraints to the model.

Forbidden Behaviour. In the previous versions of UML, it was not possible to express that, at any time, a specific scenario should not occur. However, as mentioned at the beginning of this section, a new operator called neg has been introduced for this purpose in UML 2.0. Therefore, to model what is called an anti-scenario in LSCs, we simply place it inside an interaction fragment within the scope of a neg.

3 Basic ASM Model of Sequence Diagrams

In this section we describe the basic ASM model formalizing the behaviour of sequence diagrams. Firstly, we define the signature modelling the constructs in sequence diagrams and extend them to include the concept of location. Locations are not present in UML sequence diagrams but play nonetheless a fundamental role in their semantics. In fact, the behaviour of an instance is determined according to the “nature” of the point in its lifeline it currently lies on as well as the governing execution order.

At this point, we leave out temperatures, assuming all locations are cold. Moreover, we only deal with asynchronous messages: when an instance sends a message, it can then progress along its lifeline without waiting for the message to be completed.

3.1 The Signature

Let INSTANCE be the set of instances participating in the interaction. Instances will act as agents in our model, going through their lifeline and executing what is required according to the semantics of the combined fragment they are enclosed into and the location they lie on.

Following the UML metamodel, an interaction can be a combined fragment, an interaction operand, or a state invariant. Accordingly, we define the universe INTERACTION as partitioned into COMBINED-FRAGMENT, INTERACTION-OPERAND, and STATE-INFRINGEMENT.

Combined fragments are of the form: combined-frag(optr,operands,instances), where optr: OPERATOR=\{alt,strict,seq,par,neg,assert\} represents the operator governing the fragment; operands: INTERACTION-OPERAND* is the
(ordered) sequence of operands within the fragment, and instances: \textit{INSTANCE*} is the sequence of instances involved in the fragment.

Let \textit{LOCATION} be the universe of locations in the diagram. Given an instance, we assume its locations are identified by a unique, incrementing integer along the instance’s lifeline. Therefore, we can identify \textit{LOCATION} with the set of integers \textit{INT}. In case a location is within the scope of a strict order, it will be annotated with two integers denoting, respectively, the order of the location in the instance lifeline and at the global (fragment) level. We assume that these numbers are allocated statically once the diagram is given. For example, in Fig. 1 location 4 within the \textbf{strict} combined fragment for the instance :x, is actually the third to be executed in the fragment\textsuperscript{2}.

\textbf{Fig. 1. A sequence diagram}

Locations can be of different types: \textit{begin-cf(comb-frag)} and \textit{end-cf(comb-frag)} indicate the location given by the intersection of the instance’s lifeline and the begin, respectively end, line of a combined fragment; \textit{separator(operand,comb-frag)}, indicates the line that separates two operands in a combined fragment and in particular, defines the beginning of \textit{operand}; \textit{receive(msg,inst)} and \textit{send(msg,inst)} denote the location corresponding to the point where a message is sent/received by the instance; finally \textit{state-invariant(state-constr)} is the type of a location in correspondence of a state invariant. The type of a location is given by the location

\textsuperscript{2} The last locations in Fig. 1 are not labelled as the aim of the figure is to show how locations are numbered in a sequential and strict order.
function \( \text{loc-type} : \text{INSTANCE} \times \text{LOCATION} \rightarrow \text{LOCATION-TYPE} \), where \( \text{LOCATION-TYPE} \) is the set of possible location types described above.

We define the function \( \text{next} : \text{INSTANCE} \times \text{LOCATION} \times \text{LOCATION-TYPE} \rightarrow \text{LOCATION} \) that returns the next location (number) of a certain type in the lifeline of an instance.

Let \( \text{MESSAGE} \) be the set of messages in the interaction. Messages are of the form \( \text{message}(\text{sender},\text{label},\text{receiver}) \), where \( \text{sender}(\text{message}) \) (respectively, \( \text{receiver}(\text{message}) \)) returns the instance that sends (resp. receives) the message, and \( \text{label}(\text{message}) \) returns the label of the message itself.

In case of an \( \text{alt} \) fragment, we define the function \( \text{int-constr} : \text{INTERACTION-OPERAND} \rightarrow \text{INTERACTION-CONSTRAINT} \) that returns the expression of the constraint that guards the operand.

The function \( \text{frag} : \text{INSTANCE} \times \text{LOCATION} \rightarrow \text{COMBINED-FRAGMENT} \) returns the innermost interaction fragment encompassing the location on which an instance lies. To determine the innermost interaction fragment within which an instance is executing, we define the derived function \( \text{curr-cf} : \text{INSTANCE} \rightarrow \text{COMBINED-FRAGMENT} \), as \( \text{curr-cf}(\text{inst}) = \text{frag}(\text{inst},\text{curr-loc-num}(\text{inst})) \).

Notice that we can always assume that instances are within a combined fragment, even at the top level. In fact, when left unspecified, sequence diagrams are by default regulated by a weak partial order and therefore we can consider the whole diagram within a \( \text{seq} \) combined fragment (see definition of the \( \text{seq} \) operator in section 2).

While an instance is in a location, it can receive messages from other instances. These messages must be recorded until the instance reaches their target location and executes them. Accordingly, we define the function \( \text{recMsgSet} : \text{INSTANCE} \rightarrow \mathcal{P}(\text{MESSAGE}) \) that returns the set of messages pending to be executed by a given instance\(^4\).

### 3.2 Rules

In this section we describe the rules formalizing the behaviour of instances involved in the interaction, according to the location type they currently lie onto. At any time, we need to keep track of the location the instance is required to compute. This is done by the dynamic function \( \text{curr-loc} : \text{INSTANCE} \rightarrow \text{LOCATION} \), which therefore dictates the order of execution of locations in a partial order setting. The new value for \( \text{curr-loc} \) is computed by the macro \( \text{progress} \) (see Location progress for details).

Although the UML specification does not provide any clear explanation of what happens when the end line of a \( \text{neg} \) fragment is reached, we take the LSCs approach, i.e. we assume that the system requirements are violated and the instance should therefore abort. Accordingly, we define the function \( \text{mode} : \text{INSTANCE} \rightarrow \{\text{valid},\text{aborted}\} \). As an instance is allowed to execute only if in valid mode, we assume all the rules in the model implicitly guarded by \( \text{mode}(\text{Self}) = \text{valid} \).

---

\(^3\) This \( \text{seq} \) is not to be confused with the ASM \( \text{seq} \) operator.

\(^4\) Notice that in this case the order of reception is not relevant.
If the instance lies on a location of type send message and is enabled to execute (see **Location progress** for details on the predicate enabled), then it inserts the message in the recMsgSet of the message target instance, and progresses to the next location (notice that we are dealing with asynchronous messages).

**Rule SendMessage**

\[
\text{if } \text{loc-type}(\text{Self}, \text{curr-loc}(\text{Self})) = \text{send}(m, i) \& \text{ enabled} (\text{Self}) \\
\text{then } \text{recMsgSet}(i) := \text{recMsgSet}(i) \cup \{m\} \\
\text{progress} (\text{Self})
\]

When an instance lies on a location of type receive message and is enabled to execute, it first checks whether the message has been already sent and is pending in the set of sent messages. If this is the case, the message is executed and removed from the set. The instance then progresses to the next location. This is formalized by the following rule.

**Rule ReceiveMessage**

\[
\text{if } \text{loc-type}(\text{Self}, \text{curr-loc}(\text{Self})) = \text{receive}(m, i) \& \text{ enabled}(\text{Self}) \\
\text{then if } \exists m \in \text{recMsgSet}(\text{Self}): \text{sender}(m) = i \\
\text{then } \text{recMsgSet}(i) := \text{recMsgSet}(i) \setminus \{m\} \\
\text{execute}(m) \\
\text{progress}(\text{Self})
\]

If an instance reaches the beginning line or an operand separator of an **alt** combined fragment, then if the combined fragment of the operand evaluates to true it progresses within the operand. Otherwise, it jumps to the location of the separator of the next operand in the fragment. To prevent that an instance executes more than one operand within an **alt** fragment, we set the function \( alt\text{-done: INSTANCE \times COMBINED\text{-}FRAGMENT \to BOOLEAN } \) to true as soon as one of the operands of the fragment is executed (i.e. its constraint evaluates to true).

**Rule EnterAltCombFrag**

\[
\text{if } \text{loc-type}(\text{Self}, \text{curr-loc}(\text{Self})) = \text{begin-cf(combined-frag(alt,opnd,insts))} \\
\text{then if } \text{eval(int-constr(fst(opnd)))} = \text{true} \\
\text{then } \text{progress}(\text{Self}) \\
\text{alt-done}(\text{Self}, \text{curr-loc}(\text{Self})) := \text{true} \\
\text{else } \text{curr-loc}(\text{Self}) := \text{separater(snd(opnd),combined-frag(alt,opnd,insts))} \\
\text{else if } \text{loc-type}(\text{Self}, \text{curr-loc}(\text{Self})) = \text{separator(opj,combined-frag(alt,opnd,insts))} \\
& \& \neg \text{alt-done}(\text{Self}, \text{curr-loc}(\text{Self})) \\
\text{then if } \text{eval(int-constr(opj))} = \text{true} \\
\text{then } \text{progress}(\text{Self}) \\
\text{alt-done}(\text{Self}, \text{curr-loc}(\text{Self})) := \text{true} \\
\text{else } \text{curr-loc}(\text{Self}) := \text{separator(opj+1,combined-frag(alt,opnd,insts))}
\]

If an instance reaches the begin line of a **strict**, **par** or **neg** combined fragment, then it simply progresses to the next location appropriately, as formalized below.
Observe that strict is a “shallow” operator, i.e. it defined a total order of the operands at the first level only; a strict order for nested combined fragments must be explicitly defined. This is reflected by our semantics.

**Rule EnterStrictCombFrag**

if loc-type(Self, curr-loc(Self)) = begin-cf(combined-frag(strict, opnd, insts))
then progress(Self)

**Rule EnterParCombFrag**

if loc-type(Self, curr-loc(Self)) = begin-cf(combined-frag(par, opnd, insts))
then progress(Self)

**Rule EnterNegCombFrag**

if loc-type(Self, curr-loc(Self)) = begin-cf(combined-frag(neg, opnd, insts))
then progress(Self)

If an instance reaches the end of a neg combined fragment, then it is set to mode aborted and does therefore stop its computation. Otherwise, if the instance reaches the end of any other combined fragment, it exits the fragment progressing to the next location, as formalized by the following rule.

**Rule ExitCombFrag**

if loc-type(Self, curr-loc(Self)) = end-cf(combined-frag(optr, opnd, insts))
then if optr = neg
  then mode(Self) := aborted
  else progress(Self)
where optr ∈ OPERATOR

If the instance’s current location type is state invariant, then if the invariant evaluates to true the instance can progress. Otherwise, the instance will remain in that location until the invariants holds, or the computation terminates.

**Rule StateInvariant**

if loc-type(Self, curr-loc(Self)) = state-inv(state-constr)
then if eval(state-constr) = true
  then progress(Self)

**Location Progress.** The progress macro determines the next location of the instance according to the current computation environment of the instance, i.e. whether it is in a partial, strict, or interleaving order.

As at the moment we are only dealing with cold locations, at any time the instance can actually progress or remain in the same location. This is reflected by the use of the operator choose$^5$.

---

5 We make here a use of the choose operator to select randomly between two different updates similar to what in [1] is defined as or...or.
If the instance is in a partial order setting, progressing simply means moving to the next location, i.e. incrementing the instance current location number.

\[
\text{progress}(\text{Self}) \equiv \\
\begin{cases}
\text{case curr-cf}(\text{Self}) & \\
\quad \text{strict: choose} & \\
\quad \quad \text{strict-progress}(\text{Self}, \text{curr-cf}(\text{Self})) & \\
\quad \quad \text{skip} & \\
\quad \quad \text{par: choose} & \\
\quad \quad \quad \text{interleaved-progress}(\text{Self}, \text{opnd}) & \\
\quad \quad \quad \text{skip} & \\
\quad \quad \text{else choose} & \\
\quad \quad \quad \text{curr-loc}(\text{Self}) := \text{curr-loc}(\text{Self}) + 1 & \\
\quad \quad \quad \text{skip}
\end{cases}
\]

In order to deal with strict order, we define the function \(\text{strict-curr-loc}: \text{INSTANCE} \times \text{COMBINED-FRAGMENT} \rightarrow \text{INT}\) which, at any time, returns the next location enabled to execute within the strict fragment. Therefore, to progress within a strict order, means to increment the global current location number that determines which instance is allowed to execute next. The current location of the instance is also incremented to indicated the location that will execute when it receives the global control back. We impose that \(\text{strict-curr-loc}(\text{inst}, \text{combined-frag}(\text{optr, opnd, insts})) = \text{undef}\) prior to entering the fragment.

\[
\text{strict-progress}(\text{Self}, \text{cf}) \equiv \\
\begin{cases}
\text{if strict-curr-loc}(\text{Self}, \text{cf}) = \text{undef} & \\
\quad \text{then strict-curr-loc}(\text{Self}, \text{cf}) := 1 & \\
\quad \text{else strict-curr-loc}(\text{Self}, \text{cf}) := \text{strict-curr-loc}(\text{Self}, \text{cf}) + 1 & \\
\quad \text{curr-loc}(\text{Self}) := \text{curr-loc}(\text{Self}) + 1
\end{cases}
\]

To avoid that an instance sends or receives a message within a strict environment before it is actually its turn to execute, we check that its order number is equal to the fragment strict current number. We guard the corresponding rules by the predicate

\[
\text{enabled}(\text{inst}) \equiv (\text{strict(curr-cf(inst))} \land \\
\quad \text{order}(\text{inst}, \text{curr-loc(inst)}) = \text{strict-curr-loc(curr-cf(inst)))} \\
\quad \lor \neg \text{strict(curr-cf(inst))}
\]

where the function \(\text{order}: \text{INSTANCE} \times \text{LOCATION} \rightarrow \text{INT}\) returns the order of execution of a location within a strict fragment.

Within a par fragment, the control passes from one operand to another. In order to resume the interaction from the correct point in an operand when it is (randomly) selected to progress, we define the function \(\text{par-curr-loc}: \text{INSTANCE} \times \text{INTERACTION-OPERAND} \rightarrow \text{INT}\). In case the fragment is complete, i.e. all the messages in all the operands have been executed, then the control is moved to the end of the fragment.
interleaved-progress(Self,opnd) ≡
if completed-par(Self,opnd)
then curr-loc(Self) := next(Self,c curr-loc(Self),end-cf(curr-cf(Self))
else choose op in opnd
  curr-loc(Self) := par-curr-loc(Self,op) + 1
  par-curr-loc(Self,op) := par-curr-loc(Self,op) + 1

where
completed-par(Self,opnd) ≡
∀ o ∈ opnd loc-type(par-curr(Self,o)) = separator(o,curr-cf(Self)) ∨
loc-type(par-curr(Self,o)) = end-cf(curr-cf(Self))

4 Refined ASM Model of Sequence Diagrams

In this section we refine the ASM model presented in the previous section in order to allow the distinction between hot and cold locations, messages and interactions, and to synchronise instances upon combined fragments borderlines and separators.

4.1 Adding a Temperature to the Model

We refine here the ASM model in order to deal with hot temperatures. This implies a refinement of the progress macro, to distinguish between mandatory (hot) and possible (cold) progress. Accordingly, we define the function temp : SEQUENCE × LOCATION → {hot,cold} that returns the temperature of a location in the lifeline of a certain instance. If the location on which the instance lies is hot, then its current location is updated. Otherwise, as in the basic model, the instance can randomly progress or stay in the same location.

progress(Self) ≡
case curr-cf(Self)
  strict: if temp(Self,c curr-loc(Self)) = hot
    then strict-progress(Self,curr-cf(Self))
    else choose
      strict-progress(Self,curr-cf(Self))
      skip
  par: if temp(Self,c curr-loc(Self)) = hot
    then interleaved-progress(Self,opnd)
    else choose
      interleaved-progress(Self,opnd)
      skip
else if temp(Self,c curr-loc(Self)) = hot
then curr-loc(Self) := curr-loc(Self) + 1
else choose
  curr-loc(Self) := curr-loc(Self) + 1
  skip
To formalize assert combined fragments, we simply assume that all locations within those fragments are hot. This decision comes from the resolution that it would make little or no sense at all to have a hot interaction fragment (for which the end line must be reached) and allow in it cold locations (where an instance could, in principle, remain forever).

4.2 Synchronising Instances

We have seen that the lack of synchronisation of instances when accessing or exiting combined fragments can cause many problems and inconsistencies. In this section we refine our model to impose synchronisation. This is simply achieved by defining the predicate

\[
synchronised(Self, insts) \equiv \\
\forall i \in insts, \text{loc-type}(i, \text{curr-loc}(i)) = \text{loc-type}(Self, \text{curr-loc}(Self))
\]

and adding it as a conjunct in the guard of those rules dealing with location types requiring synchronisation.

5 Conclusions and Related Work

In this paper, we give an overview of some of the most relevant and newly added features of sequence diagrams in UML 2.0. Sequence diagrams still have several problems and some aspects of their (informal) semantics are ambiguous or absent. Further, though they have been influenced by LSCs, they can still not adequately distinguish between must and may elements, necessary and possible behaviour.

In a companion paper [4], we have addressed the dichotomy of necessary and possible elements and behaviour by simply enriching a sequence diagram with constraints for liveness. The constraints are given in a simple temporal extension of UML’s constraint language OCL 2.0 [10]. The extension consists of a liveness template after eventually (essentially as introduced in [2] but more expressive in [4]). This template can be used at the local level of a message or a lifeline, and at the global level of an interaction. For example, if used at the local level of a message it expresses that after a message has been sent it will eventually be received. If used on a lifeline, it ensures that an instance must progress along its lifeline. Further, when used at the global level it enforces an interaction to occur. For the purposes of this paper, liveness at the local level is equivalent to changing the temperature of locations from cold (possible) to hot (mandatory). By default locations in a sequence diagram are cold.

Notice that in LSCs we can actually talk about the temperature not only of locations but also of messages and more generally subcharts. We believe that it is not essential to distinguish between hot and cold messages. Ultimately, what matters is that if a receive event location is hot the message must be received (which would imply that the message is hot anyway). There is also a discussion on the implications of message versus location temperatures in [7]. Concerning
hot subcharts, which in sequence diagrams are given by the operand **assert**, we can assume without loss of generality that all locations in the subchart are hot.

There are several advantages of our simple liveness extension to sequence diagrams. On the one side, we solve synchronisation issues which have been ignored in the UML 2.0 specification. On the other side, sequence diagrams are endowed with an expressiveness which is closer to LSCs. This is particularly important, because it facilitates the step of automatic code generation from inter-object requirement specifications.

In this paper, we address the semantic problems of sequence diagrams in UML 2.0 by defining a formal and rigorous semantics based on ASMs for a large subset of sequence diagrams. This builds our **basic ASM model**. The semantics of our liveness-enriched sequence diagrams is given by a simple refinement of this model in order to deal with both synchronisation and location temperature.

Consequently, by defining a semantics for sequence diagrams we had to take decisions to resolve issues in [11] that were either unclear, underspecified, ambiguous or inconsistent. One issue, for instance, concerns the intended reaction of the system to violations, i.e. invalid traces produced by an interaction fragment (**neg**) or by a single lifeline (state invariants that do not hold). It is unclear whether the intention is that the system aborts or whether the current interaction fragment is exited. We decided for the former case\(^6\).

We have also discussed problems and inconsistencies that can occur if instances are not forced to synchronise at the beginning and the end of interaction fragments, and at interaction constraints. In particular, in the case of **alt** combined fragments, although never mentioned in the specification, it appears that synchronisation **should** be guaranteed among participating instances which should block until the constraint is evaluated and the operand to be executed decided.

There are two (apparent) ways in sequence diagrams to describe that a certain sequence of communications is forbidden. On the one side, sequence diagrams offer an interaction operator called **neg** for describing forbidden behaviour, that is, whatever interactions are described within this operand, they denote an invalid trace. On the other side, we can use false **state invariants** following a sequence of interactions. According to UML, if a state invariant is false when evaluated, then the previous trace is invalid. Notice, however, that there is a problem with this option. State invariants only belong to one instance in the diagram, and consequently we are not synchronising all the instances participating in the interaction at the time that the constraint is evaluated. This may lead to a situation where the instance with the invariant progresses earlier and reaches the constraint before the other instances have done their activities. This is clearly undesirable, but it is not immediate in sequence diagrams that situations like this are or have to be prevented. Our assumption concerning state invariants in a sequence diagram is different, and implies that state invariants cannot be used to model forbidden behaviour as above (instead, the **neg** operator has to be used). We assume that when an instance reaches a location of a state invariant

---

\(^6\) To be consistent with LSCs which regard these cases as “hot violations” (cf. [7]).
it cannot progress unless the invariant evaluates to true. Notice that deadlocks are possible, because other instances within the same combined fragment cannot exit the fragment independently.

Finally, liveness-enriched sequence diagrams constitute a language for requirements specification which can be used to describe not only the expected behaviour but more importantly the actual behaviour of a system given by an inter-object behaviour model. The additional expressive power of this extension also has advantages for automatic test generation: we can specify scenarios we want the system to necessarily exhibit or avoid, and so on. Future work includes the extension of our language for test directives in [3] w.r.t. the extended sequence diagrams.

Some work on adding mandatory behaviour to UML sequence diagrams has been carried out by Haugen and Stølen [8]. However, the authors mainly give an overview of the new notation introduced in UML 2.0, describing their trace semantics using natural language. Mandatory behaviour is added to sequence and interaction overview diagrams through the introduction of a new operator called xalt, the semantics of which is never formally explained.

The importance of liveness constraints for requirements languages is also recognised for MSCs. In this context [9] introduces a new composition operator called “trigger composition” to extend the expressiveness of MSCs for liveness/progress properties. This operator is equivalent to our OCL liveness template, consequently MSCs with trigger composition can be modelled with our liveness-enriched sequence diagrams.

References


Specification and Validation of the Business Process Execution Language for Web Services*

Roozbeh Farahbod, Uwe Glässer, and Mona Vajihollahi

School of Computing Science
Simon Fraser University, Burnaby, B.C., Canada
{rfarahbo,glaesser,mvajihol}@cs.sfu.ca

Abstract. We formally define an abstract executable semantics for the Business Process Execution Language for Web Services in terms of a distributed ASM. The goal of this work is to support the design and standardization of the language. “There is a need for formalism. It will allow us to not only reason about the current specification and related issues, but also uncover issues that would otherwise go unnoticed. Empirical deduction is not sufficient.” – Issue #42, OASIS WSBPEL TC. The language definition assumes an infrastructure for running Web services on some asynchronous communication architecture. A business process is built on top of a collection of Web services performing continuous interactions with the outside world by sending and receiving messages over a communication network. The underlying execution model is characterized by its concurrent and reactive behavior making it particularly difficult to predict dynamic system properties with a sufficient degree of detail and precision under all circumstances.

1 Introduction

In this paper, we formally define an abstract operational semantics for the Business Process Execution Language for Web Services—BPEL4WS (or BPEL) [6] – in terms of a real-time distributed abstract state machine (DASM) [14], [12]. Version 1.1 of the informal BPEL language description [6], henceforth called the language reference manual or LRM, is a forthcoming industrial standard proposed by the OASIS Web Services Business Process Execution Language Technical Committee [21]. Intuitively, BPEL is an XML based formal language for modeling and design of the networking protocols for automated business processes. As such, it builds on other existing standards for the Internet and World Wide Web and, in particular, is defined on top of the service model of the Web Services Description Language (WSDL) [20]. A BPEL process and its partners are considered as abstract WSDL services that interact with each other by sending and receiving abstract messages as defined by the WSDL model for service interaction.

Our work on BPEL builds on extensive experience from semantic modeling of various industrial system design languages, including the ITU-T language SDL [11], [7], [8] and the IEEE language VHDL [4], [3]. The goal of this work is twofold. Formalization of language semantics serves two main purposes: (1) to eliminate deficien-

* Partly supported through grants from NSERC and SFU President’s Research Grant.
cies hidden in natural language descriptions, for instance, such as ambiguities, loose ends, and inconsistencies; (2) to establish a platform for experimental validation of key language attributes by making abstract operational specifications executable on real machines. For the development of BPEL, the responsible TC at OASIS lists about seventy basic issues. “There is a need for formalism. It will allow us to not only reason about the current specification and related issues, but also uncover issues that would otherwise go unnoticed. Empirical deduction is not sufficient.” – Issue #42, OASIS WSBPEL TC [21].

Formalization of language semantics based on informally specified requirements faces the non-trivial problem of ‘turning English into mathematics’. Ideally, the formal and the informal language definition should complement each other in the endeavor to sharpen requirements into specifications. That is, the formal model provides the ultimate reference whenever the clarification of subtle language issues that are difficult to articulate in plain English requires mathematical precision. In that respect, a pragmatic orientation of formal software models and their use for practical purposes such as standardization demands for a gradual formalization of the key language attributes at different levels of abstraction and with a degree of detail and precision as needed [11].

Our definition of the abstract operational semantics presented here forms a BPEL Abstract Machine and is organized into three basic layers reflecting different levels of abstraction. The top layer, called abstract model, provides an overview and defines the modeling framework comprehensively. The second layer, called intermediate model, specifies the relevant technical details and provides the full DASM model of the core constructs of the language. Finally, the third layer, called execution model, provides an abstract executable semantics of BPEL implemented in AsmL [18]. To this end, the BPEL Abstract Machine model forms a hierarchy consisting of three DASM ground models [1], [5] obtained as the result of stepwise refinements of the abstract model. The execution model is complemented by a GUI facilitating experimental validation through simulation and animation of abstract machine runs.

The paper is organized as follows. Section 2 briefly illustrates the concept of Web services architecture and gives an overview of BPEL. Section 3 introduces the abstract model, Section 4 the intermediate model, and Section 5 the execution model. In Section 6, we discuss the verification of key language properties. Section 7 concludes the paper.

2 Web Services Architecture

Several XML based Web standards have been introduced to define the Web services space and facilitate interoperability between a variety of Web applications, for instance, in e-commerce. Each of these standards targets a specific domain within the Web services space. For example, the widely used Simple Object Access Protocol (SOAP) [19] defines a standard message passing protocol, while WSDL provides a standard way of describing Web services [20].

These standards basically provide us with a structural view of Web services. They enable us to view Web services as communication endpoints which interact with each other by sending and receiving messages via a fixed collection of ports associated
with each of the communication endpoints. To this end, WSDL and SOAP support a stateless model of Web services.

The Business Process Execution Language for Web Services (BPEL) builds on top of WSDL (and indirectly also on SOAP) effectively introducing a stateful interaction model that allows to exchange sequences of messages between business partners (i.e. Web services).

In April 2003, members of OASIS\(^1\), including IBM and Microsoft among other leading companies in the e-commerce market, formed a Technical Committee \([21]\) in order to continue work on BPEL version 1.1 with the objective to establish a standardized modeling platform and language that enables and accelerates systems design and IP exchange.

### 2.1 Overview of BPEL

A BPEL process and its partners are defined as abstract WSDL services, and they use abstract messages defined by WSDL model for interaction. Figure 1 gives an overall view of the general structure of a BPEL business process document. A process is defined by specifying its partners (Web services that this process interacts with), a set of variables that keep the state of the process and an activity defining the logic behind the interaction between the process and its partners. This definition is just a template for creating business process instances. At least one start activity\(^2\) must be defined in the activity of such a template. Whenever a message arrives for a start activity, a new instance of the business process is created and starts its execution. Therefore, process creation in BPEL is always implicit.

### 2.2 Initial Example

To better understand the basic structure and some fundamental concepts of BPEL, we will provide an example: a fictitious e-Book Store. The process of buying a book from this online store is simple. A customer first sends the order to the e-Book Store. The book store then sends the order to the publisher and also sends a shipping request to a shipping company. The book store then waits to receive a callback from the shipping company and upon receiving that callback, it replies back to the customer indicating the order is received and processed successfully.

Figure 2 illustrates the structure of the interaction between publisher, shipping company, and customer for the sample business process of our e-Book Store. A business process interacts with other services through its ports, where each port is of a certain port type specifying some set of operations. Operations can be either Input-Only, Output-Only, or Input-Output.

An abstract schema of the e-Book Store business process can also be found in Fig 2. The numbers show the order in which the events occur. The BPEL process consists of 5 basic activities, two of which being executed concurrently (as indicated by identical order numbers annotating these two events).

---

\(^1\) Organization for the Advancement of Structured Information Standards.

\(^2\) A start activity is either a receive or a pick activity that is annotated with ‘createInstance = yes’ causing a new process instance being created whenever a matching message is received.
2.3 Abstract Syntax Tree

A systematic approach to capture the complete structure of a BPEL process (focusing on the relevant aspects rather than syntactical details) is its representation in the form of an attributed abstract syntax tree [11]. Many times during this project we had to refer to a precise and concise definition of the structure of a BPEL process. As the language definition in the LRM is currently lacking an abstract syntax, we defined our own abstract syntax, which is presented in [9].

2.4 Correlation

One of the main challenges in integrating Web services, and specifically business processes, is to deal with stateful interactions. Business processes normally act according to a history of external interactions. Therefore, it is necessary to keep track of the state of each business process instance. Since we have different instances of a
business process, messages need to be delivered not only to the correct port, but also to the correct instance of the business process. To ensure global interoperability and avoid implementation dependencies, the mechanism required for dynamic binding of messages needs to be defined in a generic manner rather than leaving this to the individual implementations [6].

The need for such a mechanism can be easily seen in our e-Book Store example. Each order that is sent by the customer is handled by an e-Book Store business process instance. For each order that is sent from this process instance to the publisher, there is also one business process instance at the publisher side. These pairs of process instances need to interact with each other and as a result they need to “know” each other. Therefore, there must be a mechanism to route the messages to the correct process instances. One standard approach to this problem is to carry a business token (e.g. an order number) in all transactions between e-Book Store and the publisher. In this way, all the messages that arrive for a specific process instance should carry the desired business token.

Such a mechanism is supported in BPEL by providing the ability to define a set of such correlation tokens; i.e. a set of tokens shared by all messages in a correlation group. This set is called a correlation set. Once a correlation set is initiated, the correlation tokens are identical for all the messages in that correlation group. In this way, an application-level conversation between business process instances is identified.

2.5 Activities

Activities that can be performed by a business process instance are categorized into basic activities and structured activities. Basic activities perform simple operations like receive, reply, invoke, assign, throw, terminate, wait, and empty. Structured activities impose an execution order to a collection of basic activities. It is important that structured activities can be nested. Structured activities include sequence, switch, flow, pick and while. Sequence structures a collection of activities to take place one after another. Switch provides the ability to choose among a collection of activities. Flow enables concurrent execution of a set of activities. Pick waits on a set of events for one of them to occur and executes its corresponding activity. Finally, while executes an activity repeatedly until its condition is no longer true.

2.6 Long-Running Business Process, Compensation Behavior

Business processes are meant to define the interactions between several partners that are based on certain business logic. These processes usually have long durations and include asynchronous message passing between the partners. Consequently, error handling in such an environment is not easy. It is done by compensation, i.e. “application specific activities that attempt to reverse the effects of a previous activity that was carried out as a part of larger unit of work that is being abandoned” [6, Section 13.2]. This ability of compensating exceptions in an application-specific manner enables business processes to have so-called Long-Running (Business) Transactions (LRTs). Further information on the LRTs and their formal specification is beyond the scope of this paper. Nevertheless, we have considered this concept as an important extension to the core model as is described in [9].
3 Formalization of the Web Services Architecture

We formalize the key functional attributes of the BPEL Web services architecture based on the asynchronous computation model of distributed ASMs [14]. The primary focus is on the concurrent and reactive behavior of Web services and their interaction through communication networks. This includes concurrent control structures, communication primitives, event handling mechanisms, compensation handling, and dynamic creation and termination of services. For dealing with real time aspects, we define an abstract notion of global system time and impose additional constraints on the runs defining the behavior of our BPEL abstract machine.

Logically, the architecture splits into two basically different components, namely: (1) the TCP/IP communication network, and (2) the BPEL services residing at the communication endpoints. We separate the behavior of the network from the behavior of services by decomposing our architecture model of the BPEL abstract machine into two sub-models, each of which in turn is a distributed ASM, or DASM.

In this paper, we concentrate on the service abstract machine model, whereas a network abstract machine model is defined in [12]. The composition of these two machine models is well defined by the underlying semantics of the DASM computation model. Any interaction between these models is restricted to actions and events occurring at well-identified interfaces, i.e. the ports at the communication endpoints via which services send and receive messages.

The overall organization of the BPEL abstract machine splits into three different layers as illustrated in Figure 3. The abstract model is introduced below; the complete formal model and the executable model are presented in Section 4 and Section 5.

3.1 DASM Computation Model

A DASM $M$ is defined over a given vocabulary $V$ with a program $Π_M$ and a non-empty set $I_M$ of initial states. An initial state specifies a possible interpretation of $V$ over some potentially infinite base set $X$. Intuitively, $M$ consists of a collection of autonomously operating agents from some finite set $AGENT$. This set changes dy-
namically over runs of $\mathcal{M}$ as required to model varying computational resources. The behavior of an agent $a$, in a given state $S$ of $\mathcal{M}$, is defined by the program $\text{program}_S(a)$. An agent $a$ can be terminated by resetting $\text{program}_S(a)$ to $\text{undef}$ (not representing a valid program). To introduce a new agent $b$ in state $S$, a valid program has to be assigned to $\text{program}_S(b)$.

The creation and the termination of an agent $a$ is stated by the following two operations which, at the same time, also update the (sub-)domain of agents to which $a$ belongs.

\begin{verbatim}
new a : (domain) // creates a new agent a of type (domain) and sets program(a)
stop a    // discards agent a from the domain of agents and resets program(a)
\end{verbatim}

To cope with partial updates of sets, we follow the solution proposed in [16] and use the following operations for adding/removing an element to/from a set.

\begin{verbatim}
add a to S    // Adds element a to set S
remove a from S  // Removes element a from set S
\end{verbatim}

In every state $S$ reachable from an initial state of $\mathcal{M}$, the set AGENT is well defined as follows.

$$\text{AGENT}_S \equiv \{ x \in X : \text{program}_S(x) \neq \text{undef} \}$$

The statically defined collection of all the programs that agents of $\mathcal{M}$ potentially can execute forms the distributed program $\Pi_\mathcal{M}$.

**Concurrency and Real Time.** Intuitively, the agents of $\mathcal{M}$ model the concurrent control threads in the execution of $\Pi_\mathcal{M}$. Agents interact with each other by reading and writing shared locations of global machine states. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of *partially ordered runs* [14]. Real time behavior imposes additional constraints on DASM runs ensuring that the agents react instantaneously [15]. For details see our technical report [9].

### 3.2 BPEL Abstract Model

The top layer of the BPEL abstract machine, called abstract model, provides an overview of the architecture and defines the underlying modeling framework. A BPEL document abstractly defines a Web service consisting of a collection of business process instances. A *process* instance maintains a continuous interaction with the external world (i.e., the communication network) through two interface components, called *inbox manager* and *outbox manager*, as shown in Fig. 4.

The inbox manager takes care of all the messages that arrive at the Web service. For each such message, the inbox manager is responsible to find a process instance that is waiting for that message, and assigns the message to this instance. The outbox manager, on the other hand, delivers output messages from process instances to the network. Inbox managers, outbox managers, and process instances are modeled by three different types of DASM agents. Additionally, we introduce two further agent types, *activity agents* and *handler agents*. Each process agent is responsible to execute
a single process instance; it uses dynamically created activity agents for executing complex (structured) activities. Handler agents are responsible for compensation handling or fault handling during the execution of a process instance.

\[
\text{AGENT = INBOX\_MANAGER \cup OUTBOX\_MANAGER \cup PROCESS \cup ACTIVITY\_AGENT \cup HANDLER\_AGENT}
\]

In the initial DASM state, there are only three DASM agents: the inbox manager, the outbox manager and a dummy process. This dummy process instance simplifies the method of creating new process instances. There is always one and only one such process instance waiting on its start activity. By receiving the first matching message, the dummy process instance becomes a normal running process instance and a new dummy process instance will be created automatically by the inbox manager. The DASM program given below specifies the behavior of the inbox manager agent.

\[
\text{domain MESSAGE}
\]
\[
inboxSpace: \text{INBOX\_MANAGER} \rightarrow \text{MESSAGE\_set}
\]

\[
\text{INBOX\_MANAGER\_PROGRAM} =
\]
\[
\text{if inboxSpace(self) } \neq \emptyset \text{ then}
\]
\[
\text{choose p } \in \text{PROCESS, m } \in \text{inboxSpace(self) with match(p, m) and waiting(p)}
\]
\[
\text{Assign\_Message(p, m)}
\]
\[
\text{if p = dummyProcess then}
\]
\[
\text{new newDummy : PROCESS}
\]
\[
dummyProcess := \text{newDummy}
\]

The predicate match \((p: \text{PROCESS, m: MESSAGE})\) checks whether message \(m\) can be delivered to process \(p\) or not, trying to match message type and correlation information between the waiting process and the incoming message.

In general, a BPEL program combines two different types of activities: basic activities and structured activities. Structured activities impose an execution order to a collection of basic activities. The execution of each structured activity inside a process instance is modeled by a single DASM agent of type activity agent. Figure 5 shows the control structure of DASM activity agents where one can associate one branch from the root to a leaf with each single process instance.
Below is the DASM program that abstractly specifies the behavior of process agents. In this abstract model, we do not provide the definition of Execute_Activity.

\[
\text{RUNNING\_AGENT} \equiv \text{PROCESS U ACTIVITY\_AGENT} \\
\text{startedExecution} : \text{PROCESS} \rightarrow \text{BOOLEAN} \quad \text{// Initial value: false} \\
\text{suspended} : \text{RUNNING\_AGENT} \rightarrow \text{BOOLEAN} \quad \text{// Initial value: false} \\
\text{// Tells whether the process has started its execution or not.}
\]

\[
\begin{align*}
\text{PROCESS\_PROGRAM} &= \\
\text{if}\;\neg\text{suspended}(\text{self})\;\text{then} \\
\text{if}\;\neg\text{startedExecution}(\text{self})\;\text{then} \\
\text{startedExecution}(\text{self}) &:= \text{true} \\
\text{suspended}(\text{self}) &:= \text{true} \\
\text{else} \\
\text{stop}(\text{self}) \\
\text{else} \\
\text{Execute\_Activity}(\text{activity}(\text{self}));
\end{align*}
\]

4 Complete Formal Model

By refining the abstract model of Section 3.3, we obtain the intermediate model, which provides the full DASM model of the core constructs of BPEL. The intermediate model forms the basis for deriving the executable model in Section 5.

The previous section described how a PROCESS agent executes its main activity, but we did not define Execute_Activity at that level. Following the definition in BPEL, an activity can be any of the structured or basic activities, as follows:

\[
\begin{align*}
\text{domain} & \equiv \text{REPLY} \\
\text{domain} & \equiv \text{RECEIVE} \\
\text{domain} & \equiv \text{FLOW} \\
\text{domain} & \equiv \text{SEQUENCE} \\
\text{ACTIVITY} & \equiv \text{REPLY} \cup \text{RECEIVE} \cup \text{FLOW} \cup \text{SEQUENCE} \cup ... \\
\text{// ... and all other activity domains}
\end{align*}
\]

To execute a basic activity the corresponding rule is invoked. For executing a structured activity, a new activity agent is created to handle that specific activity.
domain FLOW_AGENT
domain SEQUENCE_AGENT
domain FLOW_THREAD_AGENT
ACTIVITY_AGENT = FLOW_AGENT ∪ SEQUENCE_AGENT ∪ FLOW_THREAD_AGENT ∪ ...
// ... and all other agents for structured activities

// In the following rule, the predicates linkStatusDefined and joinCondition state // synchronization dependencies between concurrent activities. Their definition is // captured in the complete formal model by checking certain conditions before // executing an activity. For brevity, these conditions are left abstract here.

// suspended is set to true before entering this module
Execute_Activity(activity: ACTIVITY) =
  if linkStatusDefined then
    if joinCondition then
      if activity ∈ RECEIVE then
        Execute_Receive(activity)
    ... // and all other basic activities
  if activity ∈ FLOW then
    if assignedAgent(activity) = undef then
      new f : FLOW_AGENT
      assignedAgent(activity) := f
      Initialize(f, activity)
    ... // and all other structured activities
  else
    ... // JoinCondition is false. A fault (joinFailure) is thrown.
  else
    ... // There are some activities linked to this activity that have // not yet finished execution. Therefore, the activity can not be executed yet.
  where
    linkStatusDefined = ..., joinCondition = ...

In connection with structured activities, we define a function parentAgent for linking the parent agent and the subordinate activity agent. A process instance may have a number of subordinate agents that handle the structured activities inside the process instance. For each activity agent, a derived dynamic function rootProcess is defined that returns the process instance to which the agent belongs. Furthermore, the root process has to keep track of all its subordinate agents. SubordinateAgentSet is another derived dynamic function which provides the set of subordinate agents of a process instance. These functions are defined as follows.

parentAgent: ACTIVITY_AGENT → RUNNING_AGENT // Parent agent (one layer above in the creation tree) of an agent
rootProcess: RUNNING_AGENT → PROCESS // Returns the process agent to which this running agent belongs.
subordinateAgentSet: PROCESS → ACTIVITY_AGENT-set // Returns the set of activity agents that have been created and work // under control of this process.

rootProcess(a: RUNNING_AGENT) = \{ a ⊆ PROCESS, rootProcess(parentAgent(a)) : otherwise \}
subordinateAgentSet(p: PROCESS) = \{ a | a ∈ ACTIVITY_AGENT where rootProcess(a) = p \}
ParentAgent relation is maintained by calling an Initialize rule. Whenever a new activity agent is created (either in an Execute_Activity rule or inside activity agents like flow agent) the following rule is called. This rule also updates baseActivity, the activity that must be executed by this activity agent.

```
Initialize(agent: ACTIVITY_AGENT, activity: ACTIVITY) ⇔
    parentAgent(agent) := self
    baseActivity(agent) := activity
```

Going entirely through the complete formal model is outside of the space limitations of this paper. Sections 4.1 and 4.2 thus focus on two representative examples for illustrating the BPEL abstract machine model, a basic activity and a structured activity. For further details and complementary parts of the model see [9].

### 4.1 A Basic Activity: Receive

As is described in [6, Section 11.4], receive activity plays an important role for a business process both in its life cycle\(^3\) and in its service providing to partners.

In order to execute a receive activity for a given process instance, the inbox manager has to be informed that this process instance (or one of its subordinate agents) is waiting for a message. This is done by adding an inputDescriptor to the waitingForMessage set of the root process. inputDescriptor contains sufficient information about the required message and the agent that is waiting for that message. In this way the inbox manager can inspect this list and check whether any of the desired messages is received, and if so, assigns it to the matching process instance. Therefore, the agent has to wait until the inbox manager assigns a message to it. The Boolean function receiveMode is used to distinguish between the initialization mode and the waiting mode. The inputDescriptor is removed from the set as soon as a message is assigned to its corresponding activity. Thus, the agent will be informed about the assignment and can proceed with processing the message.

```
receiveMode: RUNNING_AGENT → BOOLEAN // initial value: false
waitingForMessage: PROCESS → <RUNNING_AGENT, ACTIVITY>→ set
// For each process, this set indicates the activities waiting for a message.

Execute_Receive (activity: RECEIVE) ⇔
    let inputDescriptor = <self, activity> in
    if ~receiveMode(self) then
        receiveMode(self) := true // The running agent waits to receive a message.
        add inputDescriptor to waitingSet
    else
        if inputDescriptor ∉ waitingSet then
            // Input descriptor is removed, message is received.
            receiveMode(self) := false
            suspended(self) := false // Releasing self.
        where waitingSet = waitingForMessage( rootProcess(self) )
```

\(^3\) "The only way to instantiate a business process in BPEL is to annotate a receive [or pick] activity with the createInstance attribute set to "yes"." [6, section 11.4]
4.2 A Structured Activity: Flow

A flow activity groups a set of activities and enables their concurrent execution. A flow completes when all the activities in the flow have completed [6].

For each structured activity, there is an activity agent for executing it. Flow agent is responsible for executing a flow activity. To concurrently execute the activities declared inside the flow activity, the flow agent creates a set of flow thread agents. Each flow thread agent is responsible for executing one such activity. When all the threads have finished, the flow agent releases its parent and terminates itself.

flowActivitySet: FLOW $\rightarrow$ ACTIVITY-set
// The set of all concurrent activities grouped inside a FLOW activity
flowAgentSet: FLOW_AGENT $\rightarrow$ FLOW_THREAD_AGENT-set // initial value: $\emptyset$

FLOWPROGRAM $\equiv$
if $\neg$suspended(self) then
  // Creates threads to concurrently execute activities grouped inside the flow.
  forall activity $\in$ flowActivitySet(self)
  new fThread : FLOW_THREAD_AGENT
  Initialize(fThread, activity)
  add fThread to flowAgentSet(self)
  suspended(self) := true
else
  if flowAgentSet(self) = $\emptyset$ then // All threads are done, flow activity is completed.
    suspended(parentAgent(self)) := false // The parent agent is released.
    stop self

A flow thread agent executes a single activity. Thus, its program is very similar to a process agent, except that when the execution of the activity is completed, the flow thread agent informs the flow agent by removing itself from the flow agent set.

FLOWTHREADPROGRAM $\equiv$
if $\neg$suspended(self) and $\neg$startedExecution(self) then
  startedExecution(self) := true
  suspended(self) := true
if suspended(self) then
  Execute_Activity(baseActivity(self))
if $\neg$suspended(self) and startedExecution(self) then
  remove self from flowAgentSet(parentAgent(self))
  stop self
// Each thread executes its baseActivity. When baseActivity is completed, the thread removes itself from the flow agent set as is terminated.

5 Execution Model

This section introduces an abstract executable semantics of BPEL obtained from the intermediate model as the result of another refinement step. Experimental validation of abstract requirements specifications provides us with an effective instrument to further eliminate undesirable behavior and hidden side effects already in early design stages [11]. In combination with analytical techniques, simulation and testing can provide valuable feedback for establishing key system language attributes and exploring alternative design choices. In our project, we use AsmL [18] for this purpose.
5.1 AsmL

AsmL is a rich language and its advanced language constructs are definitely helpful in rapid prototyping and object-oriented software development. For the purpose of our project, however, we have deliberately chosen a subset of the language, which is as close as possible to ‘pure ASMs.’ To facilitate modeling of the BPEL semantics, a tight relation between the full DASM model and the derived execution model is of utmost importance. Though, in order to be executable, some changes and additions were inevitable. A main weakness of AsmL is its lack of direct support for dealing with concurrency. There are no built-in constructs for simulating concurrent control threads; rather such an execution model needs to be hand coded. Ultimately, one would even expect a distributed runtime system allowing to perform truly distributed computations of DASM models encoded in AsmL.

5.2 The Model in AsmL

Intuitively, the AsmL encoding splits into four separate modules, each of which deals with a basically different aspect: (1) the original model (2) the internal environment (3) the refinement of the original model, and (4) GUI-related extensions.

The original model is basically the translation of the intermediate model to AsmL, where the main challenge was to keep it close to the pure ASMs.

The internal environment acts as an interface between our abstract machine model and the BPEL definition of the business process. In order to execute a process instance, we need a way of accessing the definition of the business process. Normally, each process instance is running an activity as defined in the BPEL process definition and determined by the history of that specific instance. One option is to encapsulate all the relevant information inside the respective entities of the model. For example, we can keep partner, port type, operation, variable and correlation sets of a receive activity inside it. Abstractly, we assumed that there is an oracle that provides this information whenever we ask for it. In the execution model, we replace this oracle with the internal environment.

In the stepwise refinement of the original model, abstract parts are refined depending on their role in the model, either by non-determinism or assigning clear deterministic behavior to them. In some cases, complex substructures had to be introduced. For example, in order to model the correlation behavior in a business process instance, we need a structure for correlation sets, mapping properties to their values. This structure completely complies with the definition of the correlation sets in BPEL. Besides, a predicate is defined to check the compatibility of a message to a correlation set, i.e. to check whether the message contains the required correlation tokens or not.

```java
class CORRELATIONSET
    var properties as map of PROPERTY to DATA
    messageContainsTokens(m as MESSAGE) as Boolean

// This method checks the compatibility of a message to a correlation set. It should check if the message carries the correlation tokens.
```
Finally, an executable model needs a GUI that makes it a useful tool for user-controlled simulation and testing. The GUI is written in Visual C# .NET\(^4\). By utilizing AsmL’s APIs with C#, we were able to integrate the model with its GUI, by defining an appropriate interface called View. For details of the execution model see [9].

5.3 Experimental Validation Results

A receive activity is a “blocking activity in the sense that it will not complete until a matching message is received by the process instance.” [6, Section 11.4]. Therefore, it is implicitly assumed that a matching message will arrive after the corresponding receive activity has been executed. Consider the following activity in a business process:

\[
<sequence>
<activity1>
<activity2>
...
<receive partnerLink="PL1" portType="PT1" operation="OP1">
</sequence>
\]

Suppose that when a process instance is executing activity2, a message arrives from partnerLink PL1, on portType PT1 and for operation OP1. Since the process instance has NOT executed the receive activity yet, it is not waiting for this message. It is not clear from the LRM what happens to such a message. Indeed, there could be multiple choices:

- **Buffer**: The message can be stored in a buffer, so that the receive activity can fetch it later.
- **Discard**: The message can simply be discarded, when there is no receive activity waiting for it.
- **Fault**: A fault can be thrown since the Web service has received a message for which no process instance is waiting.

It is certainly important for the LRM to distinguish among these choices, since it will cause inconsistencies in the behavior of different implementations of the language.

This problem was one of the problems discovered during experimental validation, when our inbox manager received a message that no process instance was expecting at the time.

6 Verification Aspects

In the current language definition, there are a number of open issues on how to establish certain key system attributes of Web services for business processes. Among those are several abstract language properties that justify formal reasoning either to prove that those properties indeed are implied by the language definition or to clarify

\(^4\) Microsoft Visual C# .NET, Microsoft Development Environment.
the resulting implicit constraints on implementations of the language, the construction of Web services, and the logic design of business processes. Two examples are discussed below.

**Correlations**

The LRM states that “After a correlation set is initiated, the values of the properties for a correlation set must be identical for all the messages in all the operations that carry the correlation set and occur within the corresponding scope until its completion” [6, Section 10.2]. Logically, the operations that carry the correlation sets can be categorized into two basically different groups: input activities, including receive, invoke, and pick, and output activities, including reply and invoke. Therefore, we can decompose the above consistency constraint into two separate constraints: (1) the property must hold on all input activities; (2) the property must hold on all output activities.

To see that the first constraint is satisfied is trivial. The LRM clearly specifies that a message must carry the required correlation tokens in order to be accepted by the process instance. This is true for every input activity. In our model, the inbox manager fulfills this duty. A message will be assigned to a process instance only if it “matches” the process instance; thus, it must carry the correlation tokens.

The second property, however, requires a closer investigation. This property can itself be decomposed to two sub-properties: (2.1) the property must hold in all output activities, where the correlation is instantiated by the same output activity; (2.2) the property must hold in all output activities where the correlation set is already instantiated.

(2.1) is confirmed by the LRM as well. The correlation set will be instantiated and the correlation tokens get their values from the message that is to be sent out. For (2.2), the language does not provide enough details to prove the second property.

In case of incoming messages, the business process is capable of filtering the messages; i.e. it will only pick those messages that match the correlation. On the other hand, in case of outgoing messages, the business process has no responsibility other than sending the message out. Although the LRM defines the semantics of a process that violates this consistency constraint as undefined, it is not precisely mentioned that output activities (like input activities) are blocking activities, and thus the loose end leads to further problems as follows.

**Synchronous Receive/Reply**

According to the LRM “A reply activity is used to send a response to a request previously accepted through a receive activity. Such responses are only meaningful for synchronous interactions.” [6, Section 11.4]. In order to clarify a request/response interaction, BPEL LRM states that “The correlation between a request and the corresponding reply is based on the constraint that more than one outstanding synchronous request from a specific partner link for a particular portType, operation and correlation set(s) MUST NOT be outstanding simultaneously.”[6, Section 11.4]. Although the definition of “outstanding” is not elucidated in the LRM, according to its interpretation by WSBPEL TC ([22, issue #26]), one can assume that an outstanding synchronous receive is a receive activity for which the required message has arrived but the reply is not sent out yet. Therefore, the following must be permissible:
<receive partnerLink="PL1" portType="P1" operation="O1" cor="C1"/>
<receive partnerLink="PL1" portType="P1" operation="O1" cor="C2"/>
<reply partnerLink="PL1" portType="P1" opr="O1">

Assuming that operation O1 is an input-output operation, these two receive activities start two synchronous request/response transactions, and as the correlation sets of these receive activities are different, these two transactions are valid to be outstanding concurrently. The problem arises when a reply message is sent to the same partner without specifying any correlation set. This is a valid reply. The problem in this case is that it is impossible to determine to which receive activity this reply is coupled; it is not clear which request/response is still outstanding and which one is not.

7 Conclusions and Future Work

Our formalization of the key semantic aspects of BPEL in terms of a hierarchically defined BPEL Abstract Machine shows that the asynchronous DASM model indeed is a natural choice for defining a precise semantic foundation. The resulting formal model transforms the abstract language definition in two consecutive refinement steps into an executable specification. In combination with inspection by analytical means, e.g. the ability to formally reason about critical language properties, experimental validation (through simulation and testing) clearly helps establishing coherence and consistence of the semantics, thereby improving the quality of the language definition. An advanced GUI facilitates such tasks (see also [9]).

A prerequisite for the feasibility of formalization when applied as a practical instrument in an industrial standardization context is conciseness, intelligibility and robustness [11]. Standardization is an ongoing and potentially open-ended activity which brings a high dynamics into the development and maintenance of a language. Such dynamics demands for a robust formalization framework that serves pragmatic needs. To this end, our abstract machine concept has already proven to be useful for enhancing conciseness and robustness of the formal model. The proposed hierarchical structuring of this model into three levels of abstraction reflects a clear separation of concerns, enhances intelligibility, and enables a tighter integration of the formal and the informal language description so that they effectively complement each other.

Our future work will concentrate on extending the BPEL Abstract Machine model towards modeling and integration of compensation behavior and fault handling.

Acknowledgements

We thank the anonymous reviewers for their valuable comments and suggestions for improvements. The idea of modeling design languages for automated business processes in terms of distributed real-time ASMs originates from joint work with Margus Veanes while one of the authors visited Microsoft Research in 2001.
References


Monodic ASMs and Temporal Verification*

Michael Fisher and Alexei Lisitsa

Department of Computer Science, University of Liverpool, Liverpool L69 7ZF, UK
{M.Fisher,A.Lisitsa}@csc.liv.ac.uk

Abstract. In this paper, we pursue the goal of automatic deductive verification for certain classes of ASM. In particular, we base our work on a translation of general ASMs to full first-order temporal logic. While such a logic is, in general, not finitely axiomatisable, recent work has identified a fragment, termed the monodic fragment, that is finitely axiomatisable and many of its subfragments are decidable. Thus, in this paper, we define a class of monodic ASMs whose semantics in terms of temporal logic fits within the monodic fragment. This, together with recent work on clausal resolution methods for monodic fragments, allows us to carry out temporal verification of monodic ASMs. The approach is illustrated by the deductive verification of FloodSet algorithm for Consensus problem, and Synapse N+1 cache coherence protocol; both are specified by monodic ASMs.

1 Introduction

The underlying idea which provides Abstract State Machines (ASMs) with much of their flexibility, is that states of an arbitrary computation are represented by general many-sorted first-order structures. The vocabularies of these structures may be freely chosen in order to model any particular algorithm at an appropriate level of abstraction [16]. (This flexibility contrasts with many standard models, such as Turing machines or finite state automata, where the alphabets used are fixed.) The process of computation is then modelled by the evolution of states according to given rules; on being ‘fired’, these rules change the structure produced by re-interpreting its basic functions.

Since its introduction, the ASM approach has proved to be very successful in specifying a variety of different types of software and hardware, including abstract algorithms, realistic programming languages, and distributed computations [1]. This success provides empirical evidence supporting the Abstract State Machine Thesis [16], which asserts that every algorithm is behaviourally equivalent to an ASM and so may be simulated, step-by-step, by that machine. For example, in [17] and [2], this thesis was established for very general formalisations of the notions of sequential and parallel algorithms, respectively. The combination of generality, varying levels of abstractness and strong formal foundations has led to the ASM approach beginning to be applied in significant industrial applications [18–20].

However, while specification and implementation techniques using ASMs have been studied in depth, there has been relatively little work on the verification of ASMs.

* The authors acknowledge partial support from EPSRC (through grants GR/M46631 and GR/R45376) for the work reported in this paper.

© Springer-Verlag Berlin Heidelberg 2004
Research in this area has, so far, mainly concerned either human-assisted proofs of correctness [36, 15], dealing with ASM representations in a relatively manual way, or algorithmic verification of ASMs [3, 9, 38, 39], based on model checking techniques [5].

In this paper, we pursue the goal of automatic deductive verification for certain classes of ASM. In particular, we base our work on a translation of general ASMs to full first-order temporal logic with natural numbers as a time flow [11]. While such a logic is, in general, not finitely axiomatisable, recent work has identified a fragment, termed the monodic fragment, that is finitely axiomatisable [23], thus making a semi-decision procedure possible. Furthermore, some of its natural (e.g., two variables or monadic) subfragments, are decidable. Thus, in this paper, we define a class of monodic ASMs whose semantics in terms of temporal logic fits within the monodic fragment. This, together with recent work on clausal resolution methods [12] for monodic fragments [6, 28, 7], allows us to carry out temporal verification of properties of monodic ASMs.

The approach is illustrated by the deductive verification of the correctness of both a FloodSet algorithm for the Consensus problem (in the presence of crash failures) and a cache coherence protocol, both specified using monodic ASMs.

We begin, in Sections 2 and 3 respectively, with descriptions of the ASM approach and the variety of temporal logic we use. In Section 4, we define how general ASMs can be translated into formulae in first-order temporal logic, and establish formal properties of such a translation. Given that a translation of general ASMs to temporal logic is possible, we next consider, in Section 5, what class of ASMs translate to monodic first-order temporal logic. We define the class of monodic ASMs that capture this property, thus allowing systems specified in this class to be translated into a temporal fragment amenable to automatic (or at least semi-automatic) verification. In Section 6, we consider two applications: one concerning a distributed consensus problem; the other concerning each coherence protocols. In Section 7 we compare our work with related work of A. Nowack. Finally, in Section 8, we provide concluding remarks and discuss future work.

2 Abstract State Machines

Definition 1
An Abstract State Machine, $\mathcal{M}$, can be defined as a tuple $\mathcal{M} = \langle \sigma, TR, IC, \Psi_0 \rangle$ where:

- $\sigma$ is a many-sorted signature, i.e. a finite collection of function names, together with their sorts; $\sigma$ is a disjoint union of three (possibly empty) parts: $\sigma = \sigma_s \cup \sigma_d \cup \sigma_e$, which consist of static, dynamic and external names, respectively. We assume the static part, $\sigma_s$ includes constants for all elements of any finite sort. We will identify boolean-valued terms with predicates whenever convenient.

- $TR$ is a finite set of transition rules, i.e. the program of $\mathcal{M}$, where a transition rule is an expression of the form
  
  if $g(\bar{x})$ then $U_1(\bar{x}) \ldots U_m(\bar{x})$

  where $g(\bar{x})$ is a first-order formula (a guard) of the given signature over variables $\bar{x}$ and $U_i(\bar{x})$ is an expression of the form $f(t_1(\bar{x}) \ldots t_k(\bar{x})) = t(\bar{x})$ where $f \in \sigma_d$ and $\bar{x}$ are all free variables of the update, although any term $t_1 \ldots t_k, t$, or guard $g(\bar{x})$ is not required to contain all of $\bar{x}$;
- $IC$ is a set of integrity constraints, with each being a first-order sentence within $\sigma_s \cup \sigma_e$.
- $\Psi_0$ is a first-order sentence in the vocabulary $\sigma$ describing initial condition.

The idea underlying the separation of $\sigma$ into three parts is as follows. The static names, i.e. those from $\sigma_s$ are assumed to not change their interpretations during the computation. The interpretations of dynamic names ($\sigma_d$) may change during the computation by the application of transition rules from $TR$. External names ($\sigma_e$) serve to model the interaction with the environment and their interpretations are not subject to change by transition rule applications, but nevertheless their interpretations may be different in different states. It is natural to consider the evaluation of an external function as being carried out by an oracle (environment).

**Note 1.** Our definition is somewhat more general than the common one, where instead of the sentence $\Psi_0$ an initial first-order structure $S_0$ is assumed.

**Note 2.** We do not allow dynamic functions within integrity constraints. There are two reasons for this. First, this would make the definition of the operational semantics more complicated. Second, one can model any possible effects of such constraints by appropriate extension of the transition rules. We feel, however, that in some cases constraints including dynamic functions are more natural and we are plan to consider these elsewhere.

**Definition 2** A state of an ASM $M = \langle \sigma, TR, IC, \Psi_0 \rangle$ is a (many-sorted) first-order structure in a signature $\sigma$.

**Definition 3** A computation (run) of an ASM $M$ is a sequence of its states $S_0, \ldots S_k, \ldots$ such that
- $S_0 \models \Psi_0$, i.e. the initial state satisfies the initial condition.
- the domain of each $S_i$ coincides with that of $S_0$;
- the interpretations of all functional symbols from $\sigma_s$ in each $S_i$ coincide with those in $S_0$;
- the interpretation of all functional symbols from $\sigma_d$ in $S_{i+1}$ is obtained from the state $S_i$ by applying of all transition rules, subject to the restrictions that every $S_i$ satisfies all the integrity constraints from $IC$ and the updates are consistent in $S_i$. The application of a transition rule within a given state $S_i$ of $M$ means simultaneously executing the updates $U_1(\bar{x}), \ldots, U_m(\bar{x})$ over the state $S_{i+1}$ whenever the guard, $g(\bar{x})$ is satisfied. Thus, if an update $f(t_1(\bar{x}), \ldots, t_k(\bar{x})) = t(\bar{x})$ is executed in $S_i$ then $S_{i+1}^1(t_1^S(\bar{x}), \ldots, t_k^S(\bar{x})) = t^S(\bar{x})$ holds for all $\bar{x}$ such that $S_i \models g(\bar{x})$. The consistency of updates in a state $S$ means that no two different updates are to update differently the interpretation of same function on the same argument.
- the interpretation of all functional symbols from $\sigma_e$ may be arbitrary, provided every $S_i$ satisfies all the integrity constraints from $IC$.

**Definition 4** If, at some state $S$, the updates are inconsistent, or execution of updates leads to the state violating some integrity constraint, it is said that $M$ crashes in $S$. For example, the rules if $b$ then $f(3) = 5$ and if $b$ then $f(3) = 7$ would lead to the crash of the machine in a state $S$ such that $S \models b$. 
Note 3. According to the Definition 3, any run is either infinite or a finite crashing one. This is essential in our approach since we will be using first-order temporal logic over an infinite flow of time isomorphic to $\omega$ to model an execution of ASMs. However, it does not prevent one dealing with finite runs of ASMs, ending by reaching some end condition (not by crashing) within this approach: one may formally extend the finite runs into infinite ones by adding obvious rules to ASM forcing the machine to loop over last state forever.

Note 4. We consider basic and restricted forms of ASM. The most notable omission is *import* instructions, allowing new elements to be imported to the domain during of computation [16]. In addition, we do not consider other extensions of ASM such as non-deterministic choice, object-orientation, or distribution mechanisms [1].

Definition 5 An ASM rule is said to be normal if, and only if, it consists of at most one update in its right-hand side. An ASM program is normal if, and only if, it consists of only normal rules.

Proposition 1. Any ASM program can be transformed into an equivalent normal ASM program.

Proof. Simply replace any rule $\text{if } g(\bar{x}) \text{ then } U_1(\bar{x}), \ldots, U_m(\bar{x})$ with the set of rules

\{ $\text{if } g(\bar{x}) \text{ then } U_i(\bar{x}) \mid i = 1 \ldots m$ \}.

3 Temporal Logic

The representation of dynamic activity via temporal formalisms is used in a wide variety of areas within Computer Science and Artificial Intelligence, for example temporal databases, program specification, system verification, agent-based systems, robotics, simulation, planning, knowledge representation, and many more [31, 32, 5, 40, 27].

3.1 First-Order Temporal Logic (FOTL)

Here, we concentrate on one very popular variety, namely discrete linear temporal logic, which has an underlying model of time isomorphic to the Natural Numbers (i.e. an infinite sequence with distinguished initial point) and is also linear, with each moment in time having at most one successor. The infinite and linear constraints ensure that each moment in time has exactly one successor.

The language $\mathcal{TL}$ of the first order temporal logic over the Natural Numbers is constructed in the standard way from a classical (non-temporal) first order language $\mathcal{L}$ and a set of future-time temporal operators ‘◊’ (sometime), ‘☐’ (always), ‘☐’ (in the next moment), ‘U’ (until).

Formulae in $\mathcal{TL}$ are interpreted in *first-order temporal structures* of the form $\mathcal{M} = \langle D, I \rangle$, where $D$ is a non-empty set, the domain of $\mathcal{M}$, and $I$ is a function associating with every moment of time $n \in \mathbb{N}$ an interpretation of predicate, function and constant symbols of $\mathcal{L}$ over $D$. First-order (nontemporal) structures corresponding to each point of time will be denoted $\mathcal{M}_n = \langle D, I(n) \rangle$. Intuitively, the interpretations of $\mathcal{TL}$-formulae
are sequences of first-order structures, or states of $\mathcal{M}$, such as $\mathcal{M}_0, \mathcal{M}_1, \ldots, \mathcal{M}_n, \ldots$. An assignment in $D$ is a function $a$ from the set $L_v$ of individual variables of $L$ to $D$. If $P$ is a predicate symbol then $P^{I(n)}$ (or simply $P^n$ if $I$ is understood) is the interpretation of $P$ in the state $\mathcal{M}_n$. We require that (individual) variables and constants of $TL$ are rigid, that is neither assignments nor interpretations of constants depend on the state in which they are evaluated. On the other hand we allow both predicates and functional symbols to be flexible. Flexibility of functions, which is not a very common assumption in the semantics of FOTL, is very convenient for direct modelling of ASM rules.

The truth-relation $\mathcal{M}_n \models^a \varphi$ (or simply $n \models^a \varphi$, if $\mathcal{M}$ is understood) in the structure $\mathcal{M}$ for the assignment $a$ is defined inductively in the usual way under the following semantics of temporal operators:

$$n \models^a \varphi \iff n + 1 \models^a \varphi;$$
$$n \models^a \Diamond \varphi \iff \text{there is } m \geq n \text{ such that } m \models^a \varphi;$$
$$n \models^a \Box \varphi \iff m \models^a \varphi \text{ for all } m \geq n;$$
$$n \models^a \varphi U \psi \iff \text{there is } m \geq n \text{ such that } m \models^a \psi \text{ and } k \models^a \varphi \text{ for every } n \leq k < m;$$

A formula $\varphi$ is said to be satisfiable if there is a first-order structure $\mathcal{M}$ and an assignment $a$ such that $\mathcal{M}_0 \models^a \varphi$. If $\mathcal{M}_0 \models^a \varphi$ for every structure $\mathcal{M}$ and for all assignments then $\varphi$ is said to be valid. For a closed formula $\varphi$ denote by $\text{Mod}(\varphi)$ the set of all its models, i.e. first-order temporal structures $\mathcal{M}$ such that $\mathcal{M}_0 \models \varphi$.

### 3.2 Monodic FOTL

While first-order temporal logic (FOTL) is very expressive and, in general, highly undecidable, recent results have identified recursively enumerable fragments of FOTLs [23].

**Definition 6** A first-order temporal formula is said to be monodic if, and only if, any subformula with its main connective being a temporal operator has at most one free variable.

The monodic fragment of FOTL has appealing properties. It is axiomatizable [41] and many of its subfragments, such as the two-variable or monadic cases, are decidable. Decidability of these fragments holds also for the case of semantics where only temporal structures over finite domains are allowed. However, these good properties hold mainly for the case when equality is not allowed. Adding equality makes even very restricted fragments non-axiomatizable [8]. A notable exception is the packed monodic fragment with equality, which is decidable [22].

### 3.3 Temporal Tools

The widespread use of temporal logics has led to the development of key techniques for utilising temporal specifications, such as model checking and deductive verification [30, 12]. While the former has restrictions in terms of the size of models considered, the latter has problems in terms of worst-case complexity. For propositional (discrete, linear) temporal logics, the decision procedure is already PSPACE [37]. In spite of this, a range
of useful tools have been developed, with our clausal resolution approach [12] having been shown to be particularly effective [26].

For general (again, discrete and linear) FOTL, incompleteness is a problem. While there have been formalisations of proof mechanisms, for example infinitary sequent systems, proof in such complex logics will require significant user interaction. We have developed a prover, based on the λClam proof planning system [35] which incorporates techniques from the clausal resolution method in order to aid proof search [4].

The breakthrough concerning monodic FOTL is now leading to the development of improved proof methods. We have extended our clausal resolution approach to this fragment [6, 28, 7], and have also provided usable implementations [24]. We believe this to be a most productive route for temporal verification in the future.

4 From ASM to FOTL

In this paper, FOTL provides the key tool for formalising the execution of ASMs. In particular, as the operational semantics of an ASM is defined in terms of runs (sequences, or partially-ordered sets, of states), these can naturally be seen as the models of FOTL.

$$[[\mathcal{M}]] = [[\Psi_0]]_{initial} \land [[\sigma]]_{sorts} \land [[\sigma]]_{static} \land [[T_R]]_{rules} \land$$

$$\land [[I_C]]_{integrity} \land [[T_R]]_{frame}$$

$$[[\Psi_0]]_{initial} = [[\Psi_0]]_{fo}$$

$$[[\sigma]]_{static} = \Box \land \bigwedge_{f \in \sigma_s} \forall \bar{x}, y \ Persist(s(f(\bar{x}) = y))$$

$$[[\{r_1, r_2, \ldots, r_n\}]]_{rules} = \bigwedge_{i=1}^n [[r_i]]_{rule}$$

$$[[\{c_1, c_2, \ldots, c_n\}]]_{integrity} = \Box \land \bigwedge_{i=1}^n [[c_i]]_{fo}$$

Fig. 1. ASM → FOTL Translation Functions

4.1 Translation

We assume in this section that an ASM program is given in a normal form. In Fig. 1, we present translation functions which effectively provide a semantics in terms of first-order temporal logic for an ASM, ‘\(\mathcal{M}\)’. In the Figure and below we use the notation \(\text{Persist}(\varphi(\bar{x}))\) to denote \(\varphi(\bar{x}) \rightarrow \Box \varphi(\bar{x})\). Four semantic functions remain undefined and require further explanation.

**First-Order Translation.** The \([[\ ]]]_{fo}\) function simply takes the many-sorted first order description given as an argument and produces a first-order description to be used in the temporal semantics. In the case of general ASMs considered here, this is simply the identity function. Later, however, we will see that some modification of the form of
the first-order sentences can be useful when translating from ASM notation to FOTL formulae.

**Sort Constraints.** The $[[\sigma]]_{\text{sorts}}$ function takes the vocabulary $\sigma$ and produces (closed) one-sorted first-order description of all sort constraints assumed for many-sorted functional symbols. For finite sorts, it also contains sentences enumerating explicitly the elements of these sorts.

**Translation of Rules.** Before we present the general translation of rules, we will consider an example. Translating the ASM transition rule, if $b(x)$ then $f(3,k(x)) = 2$, we have to take into account that the term $k(x)$ and the update $f(3,k(x)) = 2$ according to the semantics of ASM rules, should be evaluated in different time slices. So, a straightforward translation $\Box\forall x(b(x) \rightarrow \bigcirc f(3,k(x)) = 2)$ would be incorrect. The correct translation is

$$\Box\forall x \forall v[(b(x) \land (v = k(x))) \rightarrow \bigcirc f(3,v) = 2]$$

In general, given any rule $r$: if $g(\bar{x})$ then $f(t_1(\bar{x}),\ldots,t_k(\bar{x})) = t(\bar{x})$, we define the formula

$$\Phi_r(\bar{x},v_1,\ldots,v_k) \leftrightarrow g(\bar{x}) \land \bigwedge_{i=1\ldots k} (v_i = t_i(\bar{x}))$$

(1)

where all $v_i$ are fresh variables. Then we define the translation $[[r]]_{\text{rule}}$ of $r$ as

$$\Box[\forall \bar{x},\bar{v},z(\Phi_r(\bar{x},\bar{v}) \land z = t(\bar{x}) \rightarrow \bigcirc f(\bar{v}) = z)]$$

where $z$ is a fresh variable.

**Frame Conditions.** In addition to the transition rules and integrity constraints that are explicitly part of the ASM notation, there are implicit frame conditions. In essence, these conditions aim to ensure that any function retains its previous value on any argument if not affected by an update (or integrity constraint). Thus, the formula produced by $[[TR]]_{\text{frame}}$ attempts to capture this persistence. The approach we follow in generating this formula is based on the simple idea provided by Reiter [34], namely that all the potential updates and guards within transition rules are examined and new formulae are added to describe the behaviour if one or more of the guards are not satisfied.

Consider as an example the ASM program with the following transition rules:

- if $b(x)$ then $f(3,k(x)) = 2$
- if $c(x)$ then $f(x,5) = 7$
- if $d$ then $f(4,m(x)) = 3$

Given some state $s$ we are interested in identifying those elements $v$ and $w$ of the domain of $s$, for which $f(v,w)$ will not change its value after executing the above rules in $s$. It is clear that this is the case precisely when $\langle v,w \rangle$ is not a pair of values of arguments of $f$ in the left-hand side of any of the updates under substitution which makes the corresponding guard true. Now, for the above transition rules, we have the following frame condition (where variables $v$, $w$, and $z$ are universally quantified and the whole formula is assumed to be under “always” operator):
\[\begin{align*}
&\neg\exists x(v = 3 \land w = k(x) \land b(x)) \\
&\land \\
&\neg\exists x(x = v \land w = 5 \land c(x)) \\
&\land \\
&\neg\exists x(v = 4 \land w = m(x) \land d))
\end{align*}\]
\[\Rightarrow \text{Persist}_s(f(v, w) = z)\]

So, more generally, we define \([\mathcal{M}]_{\text{frame}}^\sigma\) as follows. For a function symbol \(f\) define \(I_f = \{i|i_r \in TR \text{ and } r_i \text{ updates } f\}\). Then, let \([\mathcal{M}]_{\text{frame}}^\sigma, f\) be
\[\Box \forall \bar{v}, z[\bigwedge_{i \in I_f} \neg\exists \bar{x} \Phi_{r_i}((\bar{x}, \bar{v})) \rightarrow \text{Persist}_s(f(\bar{v}) = z)]\]
where \(\Phi_r\) is defined above (1). Finally, we define \([\mathcal{M}]_{\text{frame}}^\sigma\) as \(\bigwedge_{f \in \sigma_d} [\mathcal{M}]_{\text{frame}}^\sigma, f\)

### 4.2 Correctness of the Translation

Here, we give the general statement of correctness of the translation of ASMs into full (i.e. including equality) FOTL. This is based on the strong correspondence between the execution sequences of ASMs, and the models of FOTL. The operational semantics of sequential ASMs is defined in terms of runs i.e. sequences of first-order structures, which allows to consider the infinite runs as the models for discrete, linear first-order logic. The restriction to infinite runs is essential since we consider FOTL over an infinite flow of time isomorphic to \(\omega\). (See also Note 3.)

**Theorem 1 (Correctness of Translation)** The set \(\text{Runs}(\mathcal{M})\) of infinite runs of an ASM, \(\mathcal{M}\), is exactly the set of models \(\text{Mod}([\mathcal{M}])\) of the formula \([\mathcal{M}]\), as defined in Fig. 1 and Section 4.1

**Proof [Outline].** Given \(\mathcal{M} = <\sigma, TR, IC, \Psi_0>\), then Fig. 1 defines
\[\begin{align*}
[\mathcal{M}] = [\Psi_0]^\sigma_{\text{initial}} \land [\sigma]^\sigma_{\text{static}} \land [TR]^\sigma_{\text{rules}} \land [IC]^\sigma_{\text{integrity}} \land [TR]_{\text{frame}}^\sigma
\end{align*}\]
Consider an arbitrary infinite sequence \(\gamma = S_0, S_1, \ldots\) of first-order structures in the vocabulary \(\sigma\). Then straightforward check of the definitions of run and translation shows that \(\gamma \in \text{Runs}(\mathcal{M})\) if, and only if, \(\gamma \in \text{Mod}([\mathcal{M}])\). See more details in [14].

### 4.3 Verification of ASM Translations

Given an ASM program \(\mathcal{M}\) one may use the above translation for verification of this program as follows: express correctness conditions by a first-order temporal formula \(\psi\) and then check whether the implication \([\mathcal{M}] \rightarrow \psi\) is a valid FOTL formula. In general, of course we have no decision, or even semi-decision procedure to check validity (or satisfiability) of such formulae. But, if the resulting formula turns out to be in a ‘good’ (e.g. monodic) fragment one can apply decision procedures, or theorem proving, for verification. Otherwise, one can apply heuristic methods to show its validity. Some such methods, derived from the resolution procedures for monodic fragments, and incomplete in general are presented in [4]. In what follows we describe restrictions on ASM programs that ensure their temporal translations are in monodic fragments.
5 From ASM to Monodic FOTL: Monodic ASMs

In this section we are looking for restrictions of ASM programs which ensure that their temporal translations fall within “good” monodic fragments. A quick glance at the defined translations reveals that non-monodic formulae may appear both in the translations of transition rules and in frame conditions.

Definition 7 Any term which is not a constant or variable is said to be complex, otherwise it is simple.

Definition 8 A functional symbol \( f \) of a sort \( S_1 \times \ldots \times S_k \rightarrow S \) within the ASM vocabulary is said to be finitely valued if, and only if, the sort \( S \) is finite.

Definition 9 A normal transition rule, \( \text{if } g(\bar{x}) \text{ then } f(t_1(\bar{x}), \ldots, t_k(\bar{x})) = t(\bar{x}) \) is called monodic if, and only if,

1. At most one free variable ranging over an infinite sort occurs both in the guard and in the update (i.e. the free variable is shared);
2. The main functional symbol \( f \) of an update has at most one infinite input sort;
3. An immediate subterm \( t_i(\bar{x}) \) of \( f(t_1(\bar{x}), \ldots, t_k(\bar{x})) \) of an infinite sort, is a variable (not a constant or complex term);
4. All functional symbols in the rule are finitely valued;

Definition 10 An ASM program is called monodic if, and only if,

- it consists of only normal monodic rules, and
- static part \( \sigma_s \) of the vocabulary consists of only finitely valued functional symbols of arities at most 1, and
- neither a guard of any rule, nor the initial condition of ASM nor integrity constraints contain equality.

Examples. Let \( S_i \) be an infinite sort, \( S_f \) be a finite sort and \( a \) be a constant of sort \( S_f \).

1. Assuming \( g : S_i \times S_i \rightarrow S_f \), then the rule \( \text{if true then } f(x,x) = a \) is not monodic (condition 2 does not hold)
2. Assume now that \( g : S_i \times S_f \rightarrow S_f \), \( k : S_i \rightarrow S_f \). Then the rule \( \text{if true then } g(x,k(x)) = a \) is monodic.

5.1 Translation of a Monodic ASM Program

Before we formulate a general statement on the translation of monodic ASM programs into temporal logic, it is instructive to consider an example of such a translation. Take the transition rule \( \text{if true then } f(x,k(x)) = a \) of the previous example and consider an ASM program \( M \) which contains this as its only rule. Assume also \( S_f = \{ a, b \} \). Then the translation \( [TR]_{\text{rules}} \) of the rule is \( \square \forall x, v(x = v \rightarrow \Diamond (f(v,k(v)) = a) \) and frame condition \( [TR]_{\text{frame}} \) is \( \forall v, w, z (\neg \exists x(v = x \land w = k(x)) \rightarrow Persists(f(v,w) = z)) \) which, after obvious simplification, become \( \square \forall x \Diamond (f(x,k(x)) = a) \) and
$$\forall v, w, z(w \neq k(v) \rightarrow \text{Persist}(f(v, w) = z))$$

Notice that the formula expressing the frame condition is not monodic: \text{Persist} is applied to the subformula with three free variables, which destroys monodicity. Further, both above translations contain functional symbols and equality, which also makes it problematic to be in any decidable, or axiomatizable, fragment of FOTL. However, replacing functional symbols with corresponding predicates and, further, using the finiteness conditions imposed by definition of a monodic ASM program, makes it possible to translate these formulae to the good monodic fragment, preserving the “meaning” of the original ASM program.

To replace functional symbols, we introduce new relational symbols (i.e. boolean valued functional symbols)

$$val_f : S_i \times S_f \times S_f \rightarrow \text{Bool};$$
$$val_k : S_i \times S_f \rightarrow \text{Bool}$$

Then we can rewrite $[[TR]]^\text{rules}_:\forall x \square \bigcirc (\exists u (val_k(x, u) \land val_f(x, u, a)))$ which is now monodic and without functional symbols. Further, using the finiteness of the sort $S_f$ we can remove the existential quantifier:

$$[[TR]]^*_\text{rules} = \forall x \square \bigcirc \left((val_k(x, a) \land val_f(x, a, a)) \lor (val_k(x, b) \land val_f(x, b, a))\right)$$

Now, rewrite the frame condition into

$$\forall v, w, z(\neg val_k(v, w) \rightarrow \text{Persist}(val_f(v, w, z)))$$

Notice, that this is still a non-monodic formula. Denote it by $\forall v, w \Psi(v, w)$. Further, using finiteness of $S_f$ we can remove variables $v$ and $w$ and quantifiers upon them, making the resulting formula monodic:

$$[[TR]]^*_\text{frame} = \Psi(a, a) \land \Psi(a, b) \land \Psi(b, a) \land \Psi(b, b)$$

To finish this translation we have to specify the \textit{functionality} property of newly introduced relational symbols. The corresponding formula $\text{Fun}_{k,f}$:

$$\forall x, y((val_k(x, a) \land val_k(x, b)) \land (val_f(x, y, a) \land val_f(x, y, b)))$$

has to be added to the integrity constrains $[[IC]]_{\text{integrity}}$. Here $|$ stands for “exclusive OR”. Notice that $[[TR]]^*_\text{rules} \land [[TR]]^*_\text{frame} \land \text{Fun}_{k,f}$ is a monodic formula without functional symbols and equality. Notice also that it models closely the behaviour of the initial translation $[[TR]]_{\text{rules}} \land [[TR]]_{\text{frame}}$ : the sets of their temporal models are the same modulo renaming $f \mapsto val_f$ and $k \mapsto val_k$. Now we can formulate a general statement

\textbf{Theorem 2} Let $\mathcal{M}$ be monodic ASM program. Then its temporal translation $[[\mathcal{M}]]$ can be further translated into monodic temporal formula $[[\mathcal{M}]]^*$ which:

- does not contain functional symbols;
- does not contain equality;
- has the same set of temporal models as $[[\mathcal{M}]]$ modulo some renaming of functional symbols into relational ones.
\textbf{Proof [Outline].} Given a translation $[\mathcal{M}]$ of an ASM program $\mathcal{M} = \langle \sigma, TR, IC, S_0 \rangle$ we perform further translation as follows. For every functional symbol $f$ of the sort $S_1 \times \ldots \times S_k \rightarrow S$ introduce a corresponding predicate symbol $val_f$ of the sort $S_1 \times \ldots \times S_k \times S \rightarrow \text{Bool}$. For all functional symbols $f_1, \ldots, f_s$ add $\text{Fun}_{f_1, \ldots, f_s}$, the formulae expressing functionality of all $val_{f_1}, \ldots, val_{f_s}$, to $[IC]_{\text{Integrity}}$. Because all of $f_1, \ldots, f_s$ are finitely valued and there are constants for all elements of finite sorts, $\text{Fun}_{f_1, \ldots, f_s}$ can be expressed without equality. For a term $f(t_1, \ldots, t_k)$ a (an atomic formula $Q(t_1, \ldots, t_k)$) introduce notations $f[t_1, \ldots, t_m; \bar{\theta}]$ ($Q[t_1, \ldots, t_m; \bar{\theta}]$), where $t_1, \ldots, t_m$ are all complex immediate subterms and $\bar{\theta}$ is a sequence of all simple immediate subterms (i.e variables or constants) of $f(t_1, \ldots, t_k)$ (of $Q(t_1, \ldots, t_k)$). Then for any complex term $f[t_1, \ldots, t_m; \bar{\theta}]$ define a formula $val_f[t_1, \ldots, t_m; \bar{\theta}](v)$ as follows. If all immediate subterms are simple then $val_f[t_1, \ldots, t_m; \bar{\theta}](v) \equiv val_f(\bar{\theta}, v)$, where $v$ is a fresh variable. Otherwise define it as $\exists v_1 \ldots v_m (val_{t_1}(v_1) \land \ldots \land val_{t_m}(v_m) \land val_f(v_1, \ldots, v_m, \bar{\theta}))$, where $v$ is a fresh variable.

Translate $[\mathcal{M}]$ into the formula in the new vocabulary, where all functional symbols $f$ are replaced by predicates $val_f$, as follows. Replace any atomic formula $Q[t_1, \ldots, t_m; \bar{\theta}]$ with $\exists v_1 \ldots v_m (val_{t_1}(v_1) \land \ldots \land val_{t_m}(v_m) \land Q(v_1, \ldots, v_m, \bar{\theta}))$. Replace any atomic formula $t = s$, where $t, s$ are complex terms, with $\exists \bar{v}(val_t(\bar{v}) \land val_s(\bar{v}))$. Replace any atomic formula $t = r$, where $t$ is complex and $r$ is simple term, with $val_t(v) |_{v=r}$ (i.e $r$ is substituted into $v$ in $val_t(v)$).

Given that $[TR]_{\text{frame}}$ and $[TR]_{\text{rules}}$ are now in the relational vocabulary, further translation is performed as follows: replace all quantifiers over finite sorts with conjunctions and disjunctions over elements of the sort. Then, the first and second conditions of the definition of monodic rules ensure that, in the resulting formula, we have no more than one free variable in subformulae under the $\bigcirc$ operator in $[TR]_{\text{rules}}$ and in the $\text{Persists}$ subformulae in $[TR]_{\text{frame}}$. It follows that the resulting formulae $[TR]_{\text{rules}}^*$ and $[TR]_{\text{frame}}^*$ are monodic.

Consider subformulae of the form $v_i = t_i(\bar{x})$ in the $\Phi_f$ subformulae of $[TR]_{\text{frame}}$ and $[TR]_{\text{rules}}$. During the above translation into relational vocabulary all equalities in such subformulae were eliminated, except the case, when $t_i(\bar{x})$ is a constant, or a variable. If it is a constant, then according to the condition 3 of the definition of monodic rules, it can be only of a finite sort and the quantification of $v_i$ over this sort can be eliminated together with equality. If it is a variable, then subformula $v_i = x$ of $[TR]_{\text{rules}}$ is replaced with $true$, the quantifier over $v_i$ is removed and all occurrences of $v_i$ are replaced by $x$. If $v_i = x$ is a subformula of $\Phi_f$ of $[TR]_{\text{frame}}$ then it is replaced with $true$ and is used to remove all occurrences of $x$ in $\Phi_f$. To illustrate last point consider as an example a formula, describing a frame condition:

$$\square \forall v, w, z (\neg \exists x (v = 3 \land w = x \land b(x)) \Rightarrow \text{Persists}(val_f(v, w, z))$$

One can eliminate $w = x$ getting an equivalent formula:

$$\square \forall v, w, z ((v \neq 3 \lor \neg b(w)) \Rightarrow \text{Persists}(val_f(v, w, z))$$

Thus after elimination of all such subformulae, we have final translation $[\mathcal{M}]^*$ of the initial $[\mathcal{M}]$ which is monodic and contains no equality. Finally, straightforward
but long argument shows that all stages of above translation preserve the “meaning” of temporal formulae and $[[M]^* \text{ and } [[M]]$ have the same set of temporal models modulo renaming of functional symbols into relational ones.

6 Where Are the Monodic ASMs?

The notion of monodic ASM presented in the previous section is fairly restricted. The area of its applicability still needs to be understood better. In this section we present two examples of realistic distributed algorithms (protocols) which can be specified and verified via approach we have presented.

6.1 FloodSet Algorithm

We consider here a variant of the FloodSet algorithm with alternative decision rule (in terms of [29], p.105) designed for solution of the Consensus problem in the presence of crash (or fail-stop) failures.

The setting is as follows. There are $n$ processes, each having an input bit and an output bit. The processes work synchronously, run the same algorithm and use broadcast for communication. Some processes may fail and, from that point onward, such processes do not send any further messages. Note, however, that the messages sent by a process in the moment of failure may be delivered to an arbitrary subset of the processes. Crucially, there is a bound, $f$, on the number of processes that may fail.

The goal of the algorithm is to eventually reach an agreement, i.e. to produce an output bit, which would be the same for all non-faulty processes. It is required also that if all processes have the same input bit, that bit should be produced as an output bit.

This protocol (adapted from [29]) is as follows.

– At the first round of computations, every process broadcasts its input bit.
– At every later round, a process broadcasts any value the first time it sees it.
– At every round the (tentative) output bit is set to the minimum value ever seen so far.

The correctness criteria for this protocol is that, eventually (actually, no later than in $f + 2$ rounds) the output bits of all non-faulty processes will be the same.

Claim. The above FloodSet algorithm can be specified (naturally) as a monodic ASM $M$. Its temporal translation $[[M]]$ and correctness condition a re in the decidable two-variable monodic (without equality) fragment of FOTL under finite domain semantics.

The detailed justification of this claim can be found in [14]. Here we present only one rule and its corresponding frame condition as examples. The vocabulary of ASMs here consists of (among others) unary predicate symbols $\text{Normal}$ (dynamic), $\text{Failure}$ (external) and $\text{Faulty}$ (dynamic). Intuitively $\text{Failure}(x) = \text{true}$ at some moment means “processor denoted by $x$ fails” at that moment. Similarly for $\text{Faulty}(x)$, $\text{Normal}(x)$ means $x$.

\footnote{The difference being that, in [29], every process knows the bound $f$ in advance and stops the execution of the protocol after $f + 2$ rounds, producing the appropriate output bit. For generality, we consider the version where the processes do not know $f$ in advance and produce a tentative output bit at every round.}
is not faulty nor in a failure state. One of the transition rules is if $Normal(x) \wedge Failure(x)$ then $Faulty(x) \wedge \neg Normal(x)$.

Our translation gives, for frame conditions on $Faulty$ (after simplification):

1. $\Box \forall x, z \neg (Normal(x) \wedge Failure(x)) \rightarrow Persists(Faulty(x))$ and
2. $\forall x, z \neg (Normal(x) \wedge Failure(x)) \rightarrow Persists(\neg Faulty(x))$

### 6.2 Cache Coherence Protocols

Another class of protocols which can be specified via monodic ASMs, and for which correctness can be reduced to the validity of monodic temporal formulae, is the class of cache coherence protocols [21]. These are designed to provide data consistency between caches of different processors in shared-memory multiprocessor systems, i.e. to ensure that any copies of the same memory block in the caches of different processors are identical. Abstracting from the low-level implementation details of read, write and synchronisation primitives, cache coherence protocols [21, 10] can be seen as describing the families of identical finite state machines (each being cache controller of a processor) together with a simple form of communication: when one machine makes a transition (an action) $a$, the the other machines are required to do complementary transition (a re-action) $\bar{a}$. Correctness can then be expressed in terms of co-occurrences some states in which, for example, the processors are allowed to change the content of their caches.

In [13] we have shown that theorem proving in monodic temporal fragments can be used for the verification of cache coherence protocols for an arbitrary number of processors. Here, we illustrate the point that natural ASM specifications of finite state based models of cache coherence protocols can be translated into temporal monodic specifications by the above translation procedure. As an example we consider “Synapse N+1” protocol which is particularly simple, consisting of only 3 states, 2 actions (transitions), Read and Write and two re-actions (complementary transitions) $\overline{Read}$ and $Write$. The full ASM specification of this protocol (for an arbitrary number of processors) together with comments on temporal translation and verification can be found in [14]. For the moment, we simply note that, in the ASM specification of that protocol, we assume

- the elements of the domain represent cache controllers of different processors;
- states of cache controllers are represented by unary predicates holding or not on the corresponding elements of the domain, e.g. $P(x)$ means “automaton $x$ is in state $P$”;
- use of only two (external) predicates $Read$ and $Write$ for actions, while reactions $\overline{Read}$ and $Write$ are expressed in terms of these two.

The ASM specification is monodic, except for one integrity constraint that uses an equality, expressing the fact that no more than one processor can act at every moment of time. The correctness conditions have a form

$\chi_1$: $\Box \forall x \forall y (\neg D(x) \vee \neg V(y))$

(no two controllers can be in states $D$ (“dirty”) and $V$ (“valid”) respectively).

$\chi_2$: $\forall x \forall y (D(x) \land D(y) \rightarrow x = y)$

(no more than one controller can be in state $D$).
In general, since the obtained temporal formulae are monodic, we must consider the effect of the use of the equality predicate. It is known that even severely restricted classes of monodic formulae with equality may have a highly undecidable validity checking problem [8]. However, as is shown in [13] the proof methods for the fragment without equality [4] may be used for the proof of the above correctness condition \( \chi_1 \) (and such like). As to the \( \chi_2 \), the equality may be abstracted away, by renaming the entire formula by a propositional letter and adding a corresponding \( \square \)-closed equivalence to the specification. After that the same procedure proves the validity.

7 Related Work

We have learned from one of the referees that in [33] Antje Nowack has suggested to use monodic fragments of FOTL to verify ASMs. In particular, he proposed the definition of guarded ASM and has shown that verification of the properties of such ASMs expressed in the guarded monodic fragment of FOTL is decidable. A reduction to the finite satisfiability problem of guarded monodic fragment is used.

It turns out that our work (done independently) is very much in spirit of [33]. We formulated restrictions on ASM which ensure that temporal translation falls into monodic fragment, which is, in general, undecidable, but finitely axiomatizable and hence semi-decidable. Technically, our definition of monodic ASM is less restrictive than that of guarded ASMs. On the other hand we can only guarantee existence of semi-decision procedure for monodic ASM (and suitable correctness conditions) as opposed to decision procedure for guarded ASMs (and suitable correctness conditions). However, checking whether the result of our temporal translation is in a known decidable fragment of FOTL is straightforward, so one can apply a decision procedure if it is the case. If the translation is outside known decidable fragment one can apply incomplete methods for verification. We have illustrated both these cases with the examples.

8 Conclusion

We have shown how to faithfully translate ASM specifications into the first-order temporal logic (FOTL) and have defined restrictions on ASM specifications (programs) which ensure that temporal translation is in a ‘good’ (monodic) fragment of FOTL. This allows us to use temporal translations for (semi-)automatic verifications of restricted ASM specifications either by decision procedures [7], or by theorem proving for restricted fragments of FOTL [4].

Design of effective automatic algorithms, where possible, is an important future development, as is an in-depth analysis of the applicability of monodic ASMs.

References

Towards an Interchange Language for ASMs

Elvinia Riccobene and Patrizia Scandurra

Dipartimento di Matematica e Informatica, Università di Catania
V.le A. Doria, 6 - 95125 Catania, Italy
{riccobene,scandurra}@dmi.unict.it

Abstract. This paper presents an interchange language for Abstract State Machine (ASM) models based upon a metamodel, called ASM-Metamodel (AsmM), reflecting the ASMs modelling constructs and semantics as described in [4]. The AsmM is intended to be an abstract (i.e. tool’s language independent) representation of ASMs related concepts (abstract machines, signatures, terms, rules, etc.) in order to get a standard interchange format for a systematic integration of a number of a loosely-coupled ASMs tools. The efficacy of the ASM-Metamodel in representing ASMs models is showed by means of some ASM specification examples taken from [4].

1 Introduction

The success of the Abstract State Machines (ASMs) as a systems engineering method able to guide the development of software and embedded hardware/software systems from requirements capture to their implementation, is nowadays widely acknowledged [4]. The increasing application of the ASM method for industrial projects has caused a rapid development of tools of various complexity and goals: tools for mechanically verifying ASM properties – using theorem proving systems or model checkers –, tools for executing ASMs – for simulation and testing purposes –. Since each tool usually covers an aspect of the whole system development process, at different steps, practitioners would like to change tools reusing information already entered. However, existing ASMs tools, usually developed by individual research groups, do not use a standard notation. They have syntaxes strictly depending on the target tool environment (compare, for example, AsmGofer [13], ASM-SL [6], XASM [3], ASML [10] for ASM models simulation) and thus, they do not allow to exchange and reuse portions of ASM specifications. Therefore, efficient tools integration and/or interaction is very difficult. The stating point of such a wide interaction is a common standardized exchange format, intended to serve as an underlying data structure for software modules of different tools.

Nowadays, tools developers attempt more and more to support an XMI (XML Metadata Interchange) [16,17] output format – a textual format built upon XML – for their applications, so to have a tool-independent medium for exchanging data. The idea presented in this paper is, therefore, that of assuming XMI as the standard interchange format among ASMs tools, and in order to provide
an XMI output format from an ASM specification, we exploit the MOF (Meta Object Facility) [1] methodology for languages’ definition through the usage of a metamodel.

We present an interchange language for ASM models based upon a MOF-compliant metamodel, called Asm-Metamodel (AsmM in brief), reflecting the ASMs modelling constructs and semantics as described in [4]. AsmM is intended to be an abstract (i.e. tool’s language independent) representation of ASMs related concepts (abstract machines, signatures, terms, rules, etc.) in a standard way.

MOF-compliant metamodels provide standard interchange formats for exchanging modelling data between tools, repositories and applications. The format is derived for any specific MOF-metamodel using the standard XMI mechanism. Therefore, ASMs tools could cooperate exchanging the XMI format of the ASM models.

Providing transformation definitions from AsmM to any platform-specific ASM model – such as, for example, an AsmGofer [13] specification – do not require applications, tools and databases be modified to arrange their internal data representations. Products-specific internals models can remain as they are. Tool authors only need to agree on the ASM metamodel describing ASMs and supply their tools with appropriate plug-in components capable to externalize (by means of XMI interchange formats) tool internal representation of ASM models. Therefore, users of existing ASM tool languages can continue to use their notations upon defining proper transformation functions from the AsmM to their languages. For example, considering an AsmGofer specification, we only need to define a precise transformation function in order to automatically map an AsmM model into an AsmGofer model. Similarly, the same procedure could be applied to an ASML [10] specification, and even we could “compile” ASMs models into concrete programming languages such as C++, C#, Java and so on, to provide efficient code generation capabilities and round trip engineering facilities as well.

The metamodeling approach can be exploited as a modular and layered way to endow a well-established methodology or modelling language with an abstract notation, discerning the abstract syntax and semantics of the modelling elements from their different concrete languages rendering which represents the elements concrete notations. We like to remark that the effort of developing a MOF-metamodel for ASM models is not greater than developing a BNF grammar for an ASM language, with the advantage of being able to derive different BNF-like grammars from the same MOF-metamodel.

The paper is organized as follows. Basic metamodeling notions are introduced in section 2, whereas in sections 2.1 and 2.2 the MOF and XMI standards are presented. In section 3, we present the ASM-Metamodel focusing, in 3.1, on its abstract syntax and, in 3.2, on its XMI interchange format. Related and future work are given in sections 4 and 5.
Towards an Interchange Language for ASMs

2 Metamodelling

In this section, we present some basic notions on metamodelling, MOF, and XMI in order to allow a reader not familiar with these concepts to be able to understand the ASM metamodel. The section can be skipped by a reader expert of metamodelling.

In the area of model-based software engineering, modelling/programming languages are defined in terms of metamodels by means of a common base formalism, called meta-language (see Fig. 1), that encompasses several methodologies defining, at the same time, common rules for inter-operability and interchange of models. A metamodel is therefore a precise definition of the constructs and rules needed for creating semantic models in a given language.

The classical framework for metamodelling is based on an architecture with four layers (see [1], Sect. 2.2.1):

\textbf{M}_0 (Data). The data (or information) layer comprises data of the real world that we wish to describe, i.e. it refers to actual instances of information.

\textbf{M}_1 (Model). The model (or metadata) layer comprises metadata that describes (in format and semantics) data in the information layer.

\textbf{M}_2 (Metamodel). The metamodel (or meta-metadata) layer comprises the description of the structure and semantics of metadata. A metamodel is an “abstract language” for describing different kinds of data, that is, a language without a concrete syntax or notation.

\textbf{M}_3 (Meta-metamodel). The meta-metamodel layer comprises the description of the structure and semantics of meta-metadata. In other words, it is the common meta-language for defining different kinds of metadata, and, therefore, metamodels.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fig1.png}
\caption{Modeling languages and meta-languages}
\end{figure}
2.1 Meta Object Facility (MOF)

The Meta Object Facility (MOF) [1] was defined by the OMG as a metaparadigm, i.e. a “model” whose instances are metamodels.

MOF is a language to define modelling languages as metamodels, and improve modelling systems in two ways. First, it gives individual modelers or modelling tools vendors the ability to extend existing metamodels or to develop new metamodels using an established and standard definition process. Second, MOF provides a standard mechanism that enables interoperability and model exchange between applications and tools that use different metamodels.

MOF resides at layer M3 of the metamodeling architecture and, as there is no higher abstraction layer, it is defined using MOF itself.

The MOF main metadata modelling constructs are:

Classes. They are type descriptions of “first class instance” MOF meta-objects [1]. Classes defined at the M2 layer (i.e. into a metamodel) have their instances (instances have object identity, state, and behavior) at the M1 layer. Structural features of classes can be described by Attributes (value holders in an instance of a class) and Operations (do not actually specify the behavior or the methods implementing that behavior, but only the name and the type signatures by which the behavior is invoked)\(^1\). Classes can inherit their structure from other classes by Generalization (the common class inheritance concept of the object-oriented paradigm).

Associations. They are the primary construct in MOF for expressing binary relationships between classes in a metamodel. At the M1 layer, a binary M2 layer association defines relationships, called links, between pairs of instances of the related classes. Each association has two AssociationEnds that may specify aggregation semantics and structural constraints on cardinality and uniqueness.

Data Types. They are introduced in MOF to represent attributes and operation parameter values that have types whose values do not have object identity. Data Types can represent primitive data types (like Boolean, Integer, String, etc.) and data type constructors allowing metamodelers to model more complex data types (like enumeration types, structure types, collection types, alias types).

Packages. They are MOF constructs to group elements (Classes, Associations and Packages themselves) into a metamodel for partitioning and modularizing the metamodel space.

Constraints. They are used to specify consistency rules, called well-formedness rules, to additionally argument metadata (or models) described by the above constructs in the metamodel. Constraint expressions are usually written in the Object Constraint Language (OCL)\(^2\) and can be evaluated respect to the metamodel to decide if models are valid.

\(^1\) Verily, Classes can also contain References, Exceptions, Constants, DataTypes, Constraints, and other elements [1].

\(^2\) OCL is a three-valued Kleene-Logic with equality for specifying constraints on graphs of object instances whose structure is described by MOF/UML class diagrams.
The above metamodeling constructs are sufficient to define the so called abstract syntax consisting of rules controlling structure and consistency of metadata (or models) at layer M_2.

We realize a MOF-compliant metamodel for the ASMs, reflecting the ASM modelling constructs and semantics as described in [4]. AsmM has not to be considered as a different kind of presentation of the ASMs, but a complementary work to [4].

A MOF metamodel-based language definition is in general articulated into the specification of:

- an abstract syntax A, i.e. the MOF-compliant metamodel plus well-formedness rules in OCL, for the definition of modelling constructs;
- a concrete syntax C in which to specify models, generally based on diagrammatic notation (shapes, connectors, layout, etc.) or textual syntax;
- the semantics S, i.e. the abstract logical space in which models, written in the given language, find their meanings.

The self-describing nature of the MOF Model has some important consequences (see [1], Sect. 2.2.2) not only to define languages, but also to enable building of tools for those languages. The MOF Model Interchange is among the additional MOF functionalities.

The MOF Model Interchange regards the definition of a stream- or file-based interchange format for M_1 models based on XML technology. Whenever a modelling language is specified in terms of a MOF-compliant metamodel, MOF enables a standard way to generate an interchange format for models in that language. This interchange format is called XMI (XML Metadata Interchange), and it is presented in the next section.

### 2.2 XML Metadata Interchange (XMI)

The XML Metadata Interchange (XMI) [16,17] is an OMG standard developed as a tool-independent medium for exchanging MOF data. It maps the MOF to the W3C’s eXtensible Markup Language (XML) [18] defining the rules how XML tags are used to represent serialized MOF-compliant models in XML.

Although it was originally intended as a means of exchanging MOF metamodels, the XMI format is independent of the level of MOF abstraction at which it is used. In XMI, a Document Type Definition (DTD) or a XML schema is defined at any M_x layer of the MOF metamodeling hierarchy, using rules appropriate for that layer. This DTD/schema can then be used to transport data at layer M_{x-1} in the form of XML documents that are consistent with the DTD/schema version, defined at layer M_x. Therefore, a DTD/schema is defined by the OMG at the M_3 for the MOF itself, and it is used to transport metamodels at layer M_2 in XML documents consistent with the MOF DTD/schema version. Similarly, a DTD/schema is defined at layer M_2 for a particular language metamodel defined in terms of MOF. This DTD/schema is then used to transport models at layer M_1 in XML format.

The capability of XML to communicate both metadata (tags) and data (element content) in a same document, enables applications to easily recover in-
stances via their metadata, making the XMI interchange format an optimal solution for interoperability in distributed heterogeneous environments.

3 Abstract State Machine Metamodel (AsmM)

In this section, we focus on syntactic aspects of the Abstract State Machine Metamodel (AsmM) presenting its abstract syntax, i.e. the MOF model for an hypothetical language for the ASMs method. This is the first step towards the definition of a complete standard language for the ASMs formalism based upon the MOF metamodelling technique. In fact, in order modelers effectively write ASMs specifications, this abstract syntax needs to be mapped in one or more equivalent concrete syntaxes. In this context, a concrete syntax is to be intended as any diagrammatic (shapes, connectors, layout, etc.), or textual (defined by means of a BNF grammar), or mixed (diagrammatic and textual) notation obtained\(^3\) from the AsmM.

Since a metamodel defines a set of concepts and the relations between these concepts, the AsmM can be used as an abstraction filter in a particular modelling activity where an ASM model of a system can be observed and represented in several environments and tools. The ASM notations used in those tools may be different but strongly related at the same time, because they all denote the same ASM concepts defined in the AsmM.

The definition of one or more equivalent concrete notation for the AsmM is an ongoing work. Here we focus only on those issues related to the AsmM abstract syntax, presenting the most significative MOF class diagrams of the AsmM. According to the MOF metamodelling technique, besides the abstract and concrete syntax, we have to provide the semantics of the language we are defining too. We intentionally skip this phase since we assume the ASMs semantics in [4] as semantics of the AsmM constructs (in their concrete or abstract appearances).

Interested people can refer to the full specification document (that will be soon released) for an in-depth description of all diagrams and OCL well-foundedness rules. In the full AsmM specification, each class defined in the metamodel is equipped with an explanation of its attributes, its associations, its semantics, its concrete syntax, and its constraints (as a set of invariants written in OCL [9]) to fix how to meaningful connect an instance of a construct to other instances, whenever this cannot be directly derived from the class diagrams.

As an immediate useful result of this approach, we provide the XMI-based model interchange format for ASM tools (see subsection 3.2 below). It has been automatically obtained as a MOF-derivative artifact through the standard MOF-to-XMI mapping specification [16,17].

3.1 AsmM Abstract Syntax

The abstract syntax defines concepts that are part of the ASM formalism using a MOF-compliant metamodel. It is organized in a unique package called Asm,

---

\(^3\) Generally, there must exist a mapping function between the concrete syntax effectively used to write models and the abstract syntax represented by the metamodel of the modelling language.
which is further decomposed in four packages. Each subpackage represents a different ASM aspect:

**AbstractStateMachines** defines the architectural constructs (modules and machines definitions) required to specify the backbone of an ASM model.

**ASMCommonDefinitions** contains all basic language constructs (terms, functions, domains, constraints, etc...) which characterize algebraic specifications.

**ASMTransitionRules** contains the definitions of all possible ASM transition rules schemes.

**DataTypes** specifies data types used to define the ASM metamodel itself, in particular to define extensions of the MOF types for class attributes. For example, the data type **FunctionKind** is an enumeration type introduced to denote in the metamodel the kind of a dynamic ASM function (*monitored, shared, controlled* and *out*). Note that these data types do not represent the type system of the AsmM at model level\(^4\).

---

4 ASMs are untyped.
Header (containing import/export clauses for functions or rules which are imported/exported from/to other ASM modules, and the signature - i.e. all the basic functions and domain declarations - proper of the ASM), a section Body (containing all definitions of functions, domains, rules and axioms\(^5\)), a set of initial states, and a compulsory additional main rule. The name of the ASM is also used as the name of the main rule. Executing an ASM means executing its main rule. Moreover, an abstract state machine can be a basic/turbo single-agent machine (see Chap. 3 and Chap. 4 of [4]), or a synchronous/asynchronous multi-agent machine (see Chap. 5 and Chap. 6 of [4]).

The above definition schema can be modelled in MOF as respectively showed in Fig. 3, 4, 5 and 6. These diagrams present the content of the AbstractStateMachines package; the elements depicted in these diagrams are to be considered the root elements from which (just like root nodes in a graph) all the other diagrams'elements (within the other packages of the metamodel) can be reached.

The key element is the class Asm (see Fig. 3). An instance of this class represents an entire ASM model of a system, namely an abstract state machine. The class Asm is endowed of an attribute name of type String representing the name of the machine, and an attribute isAsynchr of type Boolean which indicates if the machine is asynchronous multi-agent or not. Moreover, an Asm

\(^5\) Axioms express constraints one wants to assume for some functions or rules of the ASM.
instance contains a header section (instance of the class `Header`), a body section (instance of the class `Body`), an optional set of possible initial states (instances of the class `Initialization`), and an optional main rule (instance of the class `RuleDeclaration`). One of the initial states can be selected as default (as indicated by the association from the class `Asm` to the class `Initialization` terminating with the role name `default`). Note that, the composite relation between the class `Asm` (the whole) and its component classes (the parts) cited above is rendered as MOF composition (or composite aggregation) associations⁶.

An `Asm` instance without a main rule has to be considered as a library module based upon a standard module concept to syntactically structure large ASMs (as outlined in Chap. 2 of [4]). The module interface for the communication with other modules is described by import and export clauses contained in the header section of an ASM (see classes `ImportClause` and `ExportClause` on figure 4).

---

⁶ A composition or composite aggregation is a special form of binary association that specifies a whole-part relationship between the aggregate (whole) and a component part. It requires that a part instance is included in at most one composite at a time, and that the composite object is responsible for the creation and destruction of the parts. Graphically, a composition is shown by a solid filled diamond as an association end adornment.
Every ASM is allowed to use only identifiers (for functions and rules) which are defined within its header’s signature (instance of the **Signature** class) or imported from other modules. If the lists of imported identifiers from a specified ASM module are empty, the entire content of the ASM module is imported with its all functions and named rules. The imported functions will be statically included in the signature of the machine as completely new functions and the imported rules will enrich the module interface of the machine to provide further library functionalities.

The body part of an **Asm** entity is represented by an instance of the class **Body** (see figure 5), which consists of definitions of rules (instances of the class **RuleDeclaration**), definitions of static/derived functions (instances of the class **FunctionDefinition**) and concrete domains (instances of the class **Concrete-Domain**) already declared in the header section of the **ASM**\(^7\), and definitions of axioms (instances of the class **Axiom**) for functions, domains and named rules of the **ASM**.

The possible initial states of an **ASM** are modelled by the class **Initialization**. An instance of this class represents a single initial state consisting of initialization terms for concrete domains (instances of the class **DomainInitialization**) and for dynamic functions (instances of the class **FunctionInitialization**).

Some relevant **OCL Well-formedness rules** for the metamodel classes **AsmModule** and **FunctionDefinition** follow. Note that in an OCL expression, the reserved word **self** is used to refer to the contextual instance.

1. If an **Asm** instance has a main rule, the name of this rule must be equal to the name of the **Asm**:
   ```
   context: Asm
   inv: self.mainrule->notEmpty() implies self.mainrule.name = self.name
   ```

2. A function defined inside the body section of an **Asm** must be static or derived:
   ```
   context: FunctionDefinition
   inv: self.definedFunction.oclIsTypeOf(StaticFunction) or self.definedFunction.oclIsTypeOf(DerivedFunction)
   ```

3. If variables are present in the definition of a function, their number must be equal to the arity of the function:
   ```
   context: FunctionDefinition
   inv: self.variable->notEmpty() implies self.variable->size() = self.definedFunction.arity.oclAsType(Integer)
   ```

The above list of rules is not complete, but it gives an idea of how the metamodel is equipped with OCL rules. Due to the space limitation, in the remaining part of this paper we do not furthermore give other examples of OCL rules contained in the full AsmM specification.

\(^7\) A **concrete domain** is to be intended as a (usually finite, and maybe dynamic) set of the signature whose elements are specified or by an explicit initialization list or by a definition law where the “nature” of the elements of the domain is already established in terms of other existing domains (the so called **type domains**).
The **ASMCommonDefinitions Package**. It contains all basic language constructs like terms, functions, domains, axioms, rule declarations, etc., which characterize algebraic specifications.

The abstract syntax for the package is depicted in Fig. 7. Here, we briefly explain the meaning of the elements appearing in such class diagram. The hierarchy under the abstract\(^8\) class **Function** represents the notion of an ASM function from a domain to a codomain. We distinguish (see Chap. 2 of [4] for the classification of ASM functions) functions: **static** (instances of the class **StaticFunction**), **dynamic** (instances of the class **DynamicFunction**) and **derived** (instances of the class **DerivedFunction**). The further classification of dynamic functions in monitored, controlled, shared and out, is given by the value of the attribute kind of type **FunctionKind** in the class **DynamicFunction**.

The concept of ASM domain (or universe or set) is represented in the metamodel by the hierarchy under the abstract class **Domain**. We classify all possible ASM domains into the two categories **TypeDomain** and **ConcreteDomain**. The first ones represent all possible super domains and are further classified in **BasicTypeDomain** (domains for primitive data values like boolean, real, integer, natural, string, etc.), **StructuredTypeDomain** (domains for data structures over other super-domains like sets, sequences, bags, maps, tuples etc.), **AbstractTypeDomain** (abstract domains whose elements have none explicit

---

\(^8\) In the object-oriented paradigm, an **abstract class** is a class that cannot be directly instantiated; usually, objects are created from the sub-classes of the abstract class. In the diagrams abstract classes are distinguished with their word name depicted in Italics. A hierarchy of classes is modelled by generalization relationships shown as solid-line paths from the children (the more specific subclasses) to the parent (the more general superclass), with a large hollow triangle at the end of the paths where they meet the more general element.
structure), and EnumDomain to introduce a new concept of type element by an enumeration (e.g. one may define the enumeration Color = \{Red, Green, Blue\} to represent the new concept type of “color”). The second ones, instead, represent user-named subdomains of the first ones. Diagrams for the domain class model reflecting the above domain classification and domain relations are not presented here.

A term is a syntactic object interpreted in ASM states. In the metamodel, a term is an instance of one of the sub-classes of the abstract class Term, and the association-end value of the association between the class Term and the class Domain specifies the domain to which the value of the term interpretation belongs. As in first-order logic, we admit basic terms (variables, constants, function applications) modelled by the hierarchy under the abstract class BasicTerm, and in addition we introduce special terms like tuple terms, collection terms (sets, maps, sequences, bags, conditional terms), variable-binding terms (let-term, comprehension terms, quantification terms, etc.) and so on, modelled in the metamodel as subclasses of the abstract class ExtendedTerm. All detailed diagrams regarding classes populating the generalization hierarchy under the super-class Term are skipped here.

An instance of the class Axiom represents an axiom, namely a constraint that one assumes for a group of functions, rules and domains. Instead, an instance of the class Guard represents a boolean condition or formula usually appearing in rule definitions.

An instance of the class RuleDeclaration represents a macro/submachine declaration. In order to structure large ASMs, one can, in fact, define reusable units declaring \( r(v_1, \ldots, v_n) = P \), where \( v_1, \ldots, v_n \) are free variable occurring in the rule \( P \). There exist two types of reusable units, macros and submachines, each of which has a different semantics. The macros are units which are expanded with their definition whenever used, while the submachines are turbo ASMs, using turbo rules, which compress their internal subcomputations into one step. The boolean attribute isMacro in the class RuleDeclaration distinguishes between these two cases.

**The ASMTransitionsRules Package.** We classify transition rules in two groups: basic (the class hierarchy under the class BasicRule) and turbo (the class hierarchy under the class TurboRule). According to [4] the former are simply rules, like skip rule and update rule, while the latter are rules, like sequence rule and iterate rule, introduced to support practical composition and structuring principles of the ASMs. All the other rules derived from the basic or turbo rules are properly defined in a different diagram under the hierarchy of two abstract classes respectively called DerivedRule and TurboDerivedRule.

In order to render the idea how such rule constructs can be modelled in MOF, Fig. 8 and 9 show a portion of the class model for some basic transition rules.

### 3.2 ASM Model Interchange Using XMI

The main purpose of XMI (XML Metadata Interchange) is to provide an easy interchange of data and metadata between modelling tools and metadata reposi-
Towards an Interchange Language for ASMs

Fig. 8. Some basic transition rules (Part 1)

Fig. 9. Some basic transition rules (Part 2)

Itories in distributed heterogeneous environments [16]. In the ASM context, these application tools include: ASM model editors, ASM model repositories (databases for ASM models), ASM-to-Any code transformation definition editors, ASM-to-Any Code transformation definitions repositories, ASM model validators, ASM simulators, etc.

Without a common metamodel for creating and accessing data, developers must hard-wire discrete interfaces between applications in order to allow data exchange and consolidation, thereby limiting interoperability and increasing the cost of developing and maintaining heterogeneous systems.

According to the rules specified by the MOF-XMI Mapping Specification[16], an XML document type definition file, commonly named DTD, has been gen-
generated from the ASM-Metamodel. ASM models can therefore be exchanged between software tools as streams or files in the XMI standard format, and then verified with respect to the given ASM DTD.

The XMI representation format strictly depends on the ASM-Metamodel (see Fig. 8). Document generation is based on XML element containment. For each object the element start tag is generated from the object’s metaclass name, and the element attribute xmi:id provides a unique identifier for it in the entire document.

Long names of XMI elements with several dots may seem less legible than an usual textual notation adopted to write specifications; but clearly the XMI format has not to be confused with the “concrete syntax” in which modelers use to write. It has to be intended, instead, as an effective hard code to be automatically generated from the concrete visual or textual ASM specification for interchanging purposes only. An XMI-based model interchange format has several advantages respect to a pure definition of an interchange format based exclusively on XML[18]:

- only a slight knowledge of XML is required for getting the common format since the ASM DTD/schema is automatically derived from the MOF-compliant ASM-Metamodel;
- more capability to represent complex, semantically rich, hierarchical metadata;
- the graphical diagrams contained in the MOF metamodel of ASMs are more comprehensive and more legible than a purely textual XML DTD or schema because they are written in the style of the widely used and standardized object oriented notation;
- designers concentrate better their thoughts in “what to write” instead of “how to write” using a (sometimes twisted) tag-like formalism of a pure XML approach;
- several sophisticated tools already exist that are capable of producing the corresponding XMI DTD/schema automatically, which reduces the time spent on developing or updating an XML DTD significantly;
- the design process of XMI DTDs and schema is characterized by a more abstract semiformal language with clear-cut separation of technology dependent concepts from the independent concepts.

4 Related Work

In the ASM domain application, no other explicit proposals exist concerning what presented in this paper. We can only cite the work in [8] as a first attempt in this respect, but unfortunately it has never been completed. This work is a try to realize an interchange format for ASM specifications strictly dependent on those aspects typical of functional languages, and it is technically based only on the use of a pure XML [18] approach.

If we consider the definition of other special-purpose metamodels, we can mention the official ones supported by the OMG [11] for MOF itself [1], UML [15],
Towards an Interchange Language for ASMs

OCL [9], CWM [5], etc. Academic communities like the Graph Transformation community [14,2] and the Petri Net community [12,7], have also started to settle their tools on a general XMI/XML-based interchange format.

5 Conclusions and Future Directions

In this paper, we present a MOF-compliant metamodel to sketch a conceptual model to describe those aspects of ASM specifications which have to be exchanged through a common ASM base format. The metamodel has been defined as a set of class diagrams together with well-formedness rules.

The metamodel abstract syntax delivers a more readable and detailed view of the ASM design primitives for understanding the potentialities of the ASM formal method as “modelling language”, specially for those people who do not well deal with “mathematics”, but are familiar with the standard MOF. This abstract syntax could be easily mapped with all the existing ASM tool’s languages (or so called “concrete syntaxes”) providing, through the XMI base pipeline format, a uniform style of representing all characterizing parts of ASM specifications.

As future work, specific attention will be dedicated to the maintenance of the ASM metamodel to foster the use of it towards an efficient interaction among ASM tools for a higher quality design based on the ASM formalism. Furthermore, the definition of a concrete syntax in a BNF-form straightforwardly derived from the ASM metamodel is under development.

References

   http://research.microsoft.com/foundations/AsmL/  
   http://www.informatik.hu-berlin.de/top/pnml  
Specification and Implementation
Problems for C#

Nicu G. Fruja

Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
fruja@inf.ethz.ch

Abstract. During the attempts to build an Abstract State Machine (ASM) model for the semantics of C# programs, we tried to directly and faithfully reflect the intuitions and design decisions which are expressed in the C# Language Specification. This work and the comparison between the corresponding ASM models for C# and Java brought to light a few gaps and mistakes in the C# reference manual, inconsistencies with different implementations of C#. Some of these critical cases (especially the gaps) will be fully and correctly specified here by ASM rules of the model for C#.

1 Introduction

In [1], we formalize the semantics of the programming language C# by means of a mathematical model that reflects as much as possible the intuitions and design decisions underlying the language (see [2]) and which supports the programmer’s understanding of C# programs.

The use of the ASM method in our case-study aims to support the practitioner’s correct understanding of C# programs and of what can be expected when these programs run. Therefore, by using the ASM method, one goal is to clarify the dark corners in the C# reference manual, for the specification and evaluation of variations or extensions of the language, and for the mathematical study and comparison of C# and Java. The use of an ASM model allowed us 1) to express the basic C# objects and operations directly, without encoding, i.e. as abstract entities and actions, at the level of abstraction in which they are best understood and analyzed by the human reader and 2) to uncover the modular structure which characterizes the C# language and its implementation. Moreover, the ASM method allowed us in particular to specify the static and the dynamic parts of the C# semantics separately, due to the ASM classification of abstract states into a static and dynamic part.

We formally define in [1] the semantics of C# by providing an ASM model. This is an interpreter which executes arbitrary C# programs. The language specifications are presented as a sequence of six sub-languages of C# by isolating orthogonal parts of the language, and by defining an ASM (named EXECCSHARP with the corresponding index) for each of them, namely handling the imperative (C#I), procedural (C#C), object-oriented (C#O), exception handling (C#E),
function pointers (C#\textsubscript{D}) and unsafe code (C#\textsubscript{U}) features, each extending its predecessor with some new constructs. The last of them is the complete C#. Typically, a new construct introduces a number of small refinements of already given ASM rules.

Through the attempts to construct this ASM model and through the comparison between the corresponding ASM models for C# and Java [5], we discovered a few gaps in the specifications of the language C#, a situation where an optimization may go wrong and also a few mistakes in different C# implementations. Mainly, we were able to determine these problems since the dynamic semantics of the language was captured operationally by ASM rules which describe the run-time effect of program execution on the abstract state of the program. This illustrates that for the design and the mathematical analysis of a complex programming language like C#, the ASM method might be very helpful to the implementors of the language. This paper reports all these problems and their experimentation with the existing implementations of C#: .NET Framework [4], Rotor [10] or Mono [11]. In the next sections, .NET refers to .NET Framework 1.1 unless otherwise stated. Rotor which is a free, fully functional implementation of the standard for a common language infrastructure (CLI), includes also a compiler for C#. Mono 0.26 comes with an execution environment \textit{mono} which uses the Just in Time (JIT) compiler but also with an interpreted environment \textit{mint} (which does not use the JIT). When the tests are run under Mono, we will refer to both \textit{mono} and \textit{mint}. All the examples we are pointing to in this paper are listed with their outputs in the Appendix [17].

The paper is organized as follows. Section 2 exposes, mainly, an incompletely specified area in the C# reference manual [2]: types initialization. There are also shown gaps that imply the non-transitivity of the subtype relation. Within Section 3, it is proved how the .NET C# compiler violates the initialization semantics through a standard optimization technique. Section 4 exhibits several incoherences we found between the specifications and implementation, one of them, concerning the definite assignment, having being fixed in the meantime in .NET Framework 1.1 but still existing in Rotor. Section 5 sheds some light on some curious implemented issues which are nevertheless consistent with the specifications.

2 Gaps in the Specifications

2.1 The Types Initialization

One important issue of the C# language which is incompletely described in the standard [2], is the types initialization. We encountered several gaps when we tried in [1] to figure out ASM rules for features that have to do with the initialization process of a type (e.g. objects creation, reference of a constant, delegates creation). The most troubling problem we had to face and the one we will concentrate on is the timing when the classes and structs are initialized, including the special case when they have no static constructor defined. For
the structs, it is stated that the initialization process is similar to the classes initialization.

The initialization process of a class consists of executing its static constructor and the initializers for its static fields. The C# Specification [2] states in §10.11:

*The execution of a static constructor is triggered by the first of the following events to occur within an application domain:*

1. an instance of the class is created or
2. any of the static members of the class are referenced.

Note that before executing the static constructor of a class, all static fields in that class are first initialized to their default values, and then the static field initializers are executed in textual order.

The next question is when exactly a class is initialized if it does not define a static constructor. The specifications leave an amount of freedom for the initialization in such a case: the static field initializers are executed in textual order at an implementation-dependent time prior to the first use of a static field of that class ([2, §10.4.5.1]). What happens actually is the following: if we have a look at the IL code for such a class, we can easily notice that a static constructor (.cctor) containing the static field initializers is generated and inserted by the compiler and the class is marked with the attribute `beforefieldinit` (the same happens also for structs).

**Which Static Members?** One simple question which arises here is: are the references of the static members inherited from the base classes also considered members which trigger the initialization? The question seems to be natural because any of the static members declared in base classes are considered to be at the same time members of the derived class ([2, §3.4]). The answer is negative and the proof comes from the output of the example **Test1**, where we define two classes: the class `A` with one public static field `x` and the class `B` as a subclass of `A`. Each class declares explicitly its static constructor. In the `Main` method, we access `B.x` which, according to the output (where the static constructor of `A` is executed), triggers only the initialization of `A` (and not of `B` too).

Therefore, in order to be precise, the specifications should include (as Java does in [8, §12.4.1]) also the qualification `declared: any of static members declared by the class are referenced`.

**The Constants.** The comparison between the Java and C# models revealed another gap in the C# Specification. In Java, the `access` to a primitive constant (a field that is both `static` and `final`) does not trigger the initialization of the class which declares it ([8, §12.4.1]). In C#, there is a similar class member, namely the constants. Even though [2, §10.3] states that the constants are static members, their reference does not require the initialization of the class which declares them. As a proof, we have the example **Test2** that contains a class `A` which defines a constant `x`:

```csharp
public const int x = 1;
```
and a static constructor. Now, if we access the value of \( A.x \) (meaning that the event 2 above occurred), then the output shows that the static constructor of \( A \) is not executed, a fact that comes in contradiction with the reference manual. The constants are treated differently than the other static members since their values are known already from compile-time.

Therefore, in the specifications has to be explicitly stated that the references of constants declared by a class (or a struct — the proof is similar) do not trigger the initialization.

The Static Field Initializers. It is not quite clear what the specifications mean through the execution of the static field initializers prior to the first use of any of the fields. A static field initializer corresponds to an assignment ([2, \$10.4.5.1]), which sometimes cannot be executed before the use of the field, making this initialization ordering impossible. It is the case of the circular dependencies between static fields within the same class, where static fields can be observed in their default value state. If the assignments (i.e. static field initializers) were executed before the use of the fields, then it would be hopeless to notice the default state.

The missing point here, which is modeled in [1], is described below. When the static fields are initialized to their default values the class initialization state is set to InProgress (as described in the ASM model). This state becomes useful in the definition of the predicate Initialized for classes and structs:

\[
\text{Initialized}(c) \equiv (\text{typeState}(c) = \text{Initialized}) \lor (\text{typeState}(c) = \text{InProgress})
\]

The test \( \text{Initialized}(A) \) is performed also in the following class, before using \( b \) in the first initializer

```java
class A {
    static int a = b + 1;
    static int b = a + 1;
}
```

The test returns true since, at this moment \( \text{typeState}(A) = \text{InProgress} \) and consequently it is not required that the static field initializer of \( b \) be already executed (one reason could be that the initializers have to be executed in textual order). Thus the value considered for \( b \) is 0.

This explains also the tricky example Test3, where the classes \( A \) and \( B \) are defined as in Fig. 1. If we access in the Main method the fields \( A.x, A.y, B.c \) and \( B.d \) in this order, then under .NET we get for \( B.c \) the value 1. It means that the field initializer of \( A.x \) is not executed before the one of \( B.c \). This is not mandatory since before using \( A.x \) in the initializer of \( B.c \), \( \text{typeState}(A) = \text{InProgress} \) and accordingly \( \text{Initialized}(A) \) is true. If we constrain the initializer of \( A.x \) to be executed before the use of the field in \( 1 + A.x \), then \( B.c \) should be definitely 3. This is the case in Rotor and Mono.

The \textit{beforefieldinit} Attribute. In [3, Partition II, \$9.5.3.2], it is specified that if a class (or struct) is marked with \textit{beforefieldinit}, then the reference of
a static method does not trigger the class (or struct) initialization. This could mean also that if there is no reference to a static field, then the static field initializers may not be executed at all, an idea which does not exactly match the C# Specification ([2, §10.4.5.1]). Due to this underspecification of the types initialization, programs with beforefieldinit types that have static field initializers with side effects behave nondeterministically in unspecified ways.

The rational for the attribute beforefieldinit can be found in the specifications. Thus, [3, Partition II,§9.5.3.2] claims that it is expensive to ensure that type initializers are run before a static method is called, particularly when executed from multiple application domains. CLI has the feeling that this should not be a problem most of the time. But let us consider the following scenario: we have a class \( A \) which declares a field \( x \) of a class type \( B \) and defines a static method \( m \)

```csharp
class A {
    public static int x = 2;
    public static int y = B.d;
}
class B {
    public static int c = 1 + A.x;
    public static int d = 1 + A.y;
}
```

Fig. 1. The initializer of \( A.x \) is not executed before \( A.x \) is used in the initializer of \( B.c \).

Suppose that \( m \) does not reference the static field \( x \) and hence, according to the specifications, there is no need to execute the initializer of \( x \). As long as \( m \) does not depend on the initialization of \( x \) indirectly, the reasoning for beforefieldinit is all right. But what if in the process of initializing \( x \), the constructor of \( B \) sets up a static hash table in some other class upon which \( m \) depends? This means that as long as this kind of problematic side-effect does not show up, the rational for beforefieldinit is okay.

However, we can suppress the beforefieldinit by simply declaring a static constructor but it looks somehow odd for the behavior of a class (or struct) \( A \) to change (radically) just by adding `static A() {}`. This might seem to many programmers to be a no-op and it is likely to be removed.

Java requires that initialization should be lazy (i.e. triggered by the first occurrence of events 1,2). Moreover, in Java the static members should be accessed (invoked in case of methods and assigned or used in case of fields) in order to trigger the invocation of the type initializer ([8, §12.4.1]), while in C# it suffices
to have them referenced (the difference is made evident in Section 4 in the case of delegates creation). In the C# model, the classes and structs are classified using the predicate beforefieldinit, depending on whether or not they declare a static constructor. The behavior of the run-time system, which can at anytime take the decision of initializing a beforefieldinit type, is modeled in [1] through an external function typeToBeInitialized. If this function is set to such a type, then the ASM starts its initialization process (by executing the implicitly generated constructor); otherwise it runs the submachines corresponding to the 6 submodules of C#:

\[ 
\text{ExecCsharp} \equiv \\
\begin{align*}
\text{if } & \text{typeToBeInitialized} \neq \text{Undef} \text{ then} \\
& \text{initialize}(\text{typeToBeInitialized}) \\
\text{else} & \\
& \text{ExecCsharp}_I \\
& \text{ExecCsharp}_C \\
& \text{ExecCsharp}_O \\
& \text{ExecCsharp}_E \\
& \text{ExecCsharp}_D \\
& \text{ExecCsharp}_U \\
\end{align*}
\]

Moreover, in the ASM rules for static method calls and calls of instance constructor, we should check if the class is beforefieldinit since this should not trigger the initialization of the class unless this is in an erroneous state \( \text{Exc(ref)} \) (this is a technical detail which will be explained below). For example, for invoking a static method \( m \) defined in class \( c \) with a set of arguments \( v \), we created the following macro:

\[ 
\text{InvokeStatic}(c:m, v) \equiv \\
\begin{align*}
\text{if } & \neg \text{TriggerInit}(c) \text{ then } \text{InvokeMethod}(c:m, v) \\
\text{else} & \text{initialize}(c) \\
\end{align*}
\]

where \( \text{TriggerInit}(c) \) is a predicate defined by

\[ (\neg \text{Initialized}(c) \land \neg \text{beforefieldinit}(c)) \lor (\text{typeState}(c) = \text{Exc(ref)}) \]

that indicates whether this call should trigger the execution of the initialize macro (described below).

The Instance Constructors. Another problem encountered, when we were trying to compare the corresponding Java and C# models, was that the events 1,2 are not the only ones which trigger the execution of a static constructor. In Java, before a class is initialized, its superclasses must be initialized ([8, §12.4.1]), while in C# this is not the case. In C#, there is a default constructor initializer base() which is an instance constructor of the base class and has to be executed immediately before our instance constructor ([2, §10.10.1]). It seems that whenever the constructor initializer contains an instance constructor of the base class, the base class has to be initialized, triggering the execution of its
static constructor. This becomes obvious in the example **Test4**, where we define the class **A** and its subclass **B**. Each class defines explicitly its static constructor. In the **Main** method, we create an instance of **B**. The output shows that not only **B** is initialized but also **A** (after **B**). Certainly the instance constructor of **A** has to be executed before that of **B**, but why is **A** initialized? According to the specifications, there is no reason for that since no instance of **A** is created.

One explanation could be that before invoking the instance constructor of **A**, we have to assure that **A** is initialized. It follows that the invocation of an instance constructor of a class triggers the class initialization, a point which is missing from the specifications [2]. It turned out in [1], that the ASM rule for invoking an instance method is different from the corresponding one in the Java model [5]. Thus, the rule for calling an instance method **m** declared in **t**, on an instance **val**, with the list of arguments **vals** (i.e. executing **val.t::m vals**), checks in the case of an instance constructor, whether the containing class must be initialized. If this is the case, then the class is going to be initialized otherwise the method **m** is invoked unless the instance **val** is **null**, in which case a **NullReferenceException** has to be thrown:

```
if InstanceCtor(m) \&\& TriggerInit(t) then
  Initialize(t)
elseif val \neq null then InvokeInstance(t::m, val, vals)
else FailUp(NullReferenceException)
```

Note that this initialization test just before invoking the instance constructor is consistent also with the following slight difference between C# and Java. In Java, classes are initialized when **new** is encountered (this means even before evaluating any argument expressions), while in C#, classes are initialized after the argument expressions of the constructor are evaluated. One can confirm the above stated difference with the simple example **Test5**: the class **A** declares a private field **x**, an instance constructor which sets **x** and a static constructor. In the class **Test**, there are a static method **F** and the **Main** method consisting of the following object creation:

```c
A a = new A(F());
```

The output of the C# code shows that the method **F** is executed before the static constructor of **A**, while the same program written in Java yields the inverse order.

**The Default Instance Constructor of a Struct.** The next gap concerns the initialization of structs. It showed up when we tried to figure out whether the static constructors defined by classes, respectively structs behave analogously. Thus, it turned out that the creation of a struct instance using the parameterless default instance constructor does not trigger the execution of the *explicitly defined* struct static constructor, a fact which is missing from the specifications ([2, §11.3.8]) and can be certified through a very simple example (like **Test6**).
Erroneous State of a Class. Let us suppose now that some event (of type 1 or 2) determined the static constructor of a class to be executed and the execution (of the constructor or static field initializers) terminates with an exception (i.e. there was no catch clause to handle it). Then, according to [2, §16.3], a TypeInitializationException (in Java, this corresponds to the ExceptionInInitializerError) is thrown at the point that triggered the invocation of the static constructor. What is not stated at all in the specifications (and it is available also for structs), is what happens if, after we handle the TypeInitializationException, we try to access once again a static member declared in that class.

In Java, in this situation, the confusing NoClassDefFoundError is thrown. This is expressed in [5] by the extension in Java#C of the rule INITIALIZE defined in Java#C:

\[
\text{INITIALIZE}(c) \equiv \\
\ldots \\
\text{if classState}(c) = \text{Unusable then Fail(NoClassDefFoundError)}
\]

In C#, the old TypeInitializationException object is thrown again. One can sustain this idea using the example Test7 which consists of a class A that declares a public static field x and a static constructor which first sets a local integer variable i to 0 and then assigns 1/i to x. Let us assume now that in the Main method we declare a local variable e of type Exception (initialized with null) where the first TypeInitializationException object is going to be stored. We can cause this exception to be thrown if we access the value of A.x (within a try-catch block). We catch it in the catch clause and we store it into e. We consider then another try-catch block, independent of the previous, where we access again A.x. After we catch the exception (say y), we test if y is equal to e (the first thrown exception) and we get a positive answer (under .NET and Rotor). Moreover, we notice from the output that the static constructor of A did not have to start executing again in order to throw the TypeInitializationException. Mono does not seem to like this example very much: mono says that it has encountered a problem and needs to close, while mint complains that the DivideByZeroException from the static constructor is not handled.

It means that, in this kind of situation, we may consider the class in a special (erroneous) state \( \text{Exc}(\text{ref}) \) where \( \text{ref} \) is the exception that was thrown in the type initializer. This is expressed in [1] by the ASM rule for executing a node consisting of an exception, \( \text{Exc}(\text{ref}) \). If the exception reached the root position of the current method body \( (\text{pos} = \text{body}(\text{meth})) \) and there are still frames on the stack and moreover, if the current method is a static constructor \( (\text{StaticCtor}(\text{meth})) \), then the class defining it gets into an erroneous state \( \text{Exc}(\text{ref}) \), otherwise the exception \( \text{ref} \) together with the topmost frame method are added to the stack trace:
\( \text{Exc(ref)} \rightarrow \\
\text{if } pos = \text{body(meth)} \land \neg \text{Empty(frames)} \text{ then} \\
\quad \text{if } \text{StaticCtor(meth)} \text{ then} \\
\qquad \text{typeState(type(meth))} := \text{Exc(ref)} \\
\text{else} \\
\quad \text{AppendStackTrace(ref, meth(top(frames)))} \\
\quad \text{ExitMethod(Exc(ref))} \\
\)

Therefore, for C\#\varepsilon, the macro \texttt{INITIALIZE} defined in C\#\varepsilon is refined to re-throw the old exception object, preventing in this way the re-initialization.

\[
\text{INITIALIZE}(c) \equiv \\
\quad \ldots \\
\quad \text{if } \text{typeState}(c) = \text{Exc(ref)} \text{ then } \text{YIELD(Exc(ref))}
\]

The predicate \textit{TriggerInit} was defined and used (see the macro \texttt{INVOKESTATIC} and the rule for invoking an instance method) in such a way that if the class is in an erroneous state, the macro \texttt{INITIALIZE} is executed in order to throw the exception even if the class is \textit{beforefieldinit}.

### 2.2 The Interfaces Implementation

Another gap of [2] is concerning the \textit{interfaces implementation} definition provided by §13.4. There are only two cases stated for a class \(A\) to implement an interface \(I\): the base class list of \(A\) includes \(I\) or an interface \(J\) which has as a base interface \(I\) (not necessarily direct). The following natural case is missing from specifications: a class \(A\) implements an interface \(I\) also in the particular situation that \(A\) is a subclass of \(B\) and \(B\) implements the interface \(I\). This gap has as a consequence the non-transitivity of the \textit{standard implicit conversion} relation (subtyping) defined in the standard [2, §6.3.1].

On the other hand, in order to fix the non-transitivity, one should add to the specifications [2, §6.1.4] also the following implicit reference conversions:

- \textit{from any array type} to \texttt{System.Array} \textit{and to the interfaces implemented by} \texttt{System.Array}.
- \textit{from any delegate type} to \texttt{System.Delegate} \textit{and to the interfaces implemented by} \texttt{System.Delegate}.

Otherwise the subtype relation is not transitive: for example, there is an implicit reference conversion from an array type \(T\) to \texttt{System.Array} and from \texttt{System.Array} to \texttt{System.ICloneable} but there is no conversion (according to the specifications) from the array type \(T\) to \texttt{System.ICloneable}. This would mean that the assignment

\[
\text{ICloneable } x = \text{new } \text{int}[1]
\]

should be rejected at compile-time, which is not the case.
3 Incorrect Compiler Optimization

In this section, we will show how, in a special kind of assignments, the .NET C# compiler (more precise the JIT compiler) does not preserve the C# semantics defined by the specifications, by performing an illegal optimization.

The incorrect optimization is related to the timing of a null check for \( \text{exp}_1 \) in a simple assignment of the form \( \text{exp}_1.f = \text{exp}_2 \), where \( \text{exp}_1 \) is an expression of a class type \( c \) and a member lookup of \( f \) in \( c \) produces a match. Let us consider the example Test8, where class \( A \) declares the instance field \( x \) and a static constructor. Now suppose that in the Main method, we have an instance \( a \) of type \( A \) initialized with \( \text{null} \), a local variable \( i \) initialized with \( 0 \), and an assignment like in Fig. 2.

The Analysis of the Problem. The run-time processing of a simple assignment described in [2, §7.13.1], has the following first step: \( a.x \) is evaluated to produce the variable. The evaluation to produce a variable (of \( a.x \)) is not defined by the specifications, but its meaning could be more or less derived from the evaluation of a member access described in [2, §7.5.4]: on the assumption that \( a \) is a variable and a member lookup of \( x \) in \( A \) produced a match (as in our case), the result (of the evaluation) is a variable, namely the field \( x \) in the object referenced by \( a \). So we could almost claim that, this is the way, \( a.x \) can be evaluated to produce a variable. But, in the processing above, there is a first step consisting of a null check for \( a \). This would mean that a NullReferenceException should be thrown even before evaluating the right-hand side of the assignment. It is done in the same way also in Java: if the evaluation completes abruptly, then the assignment expression completes abruptly for the same reason; the right-hand operand is not evaluated and no assignment occurs ([8, §15.26]).

On the other hand, we could consider another definition for evaluated to produce the variable much closer to the IL code corresponding to the assignment (see Fig. 2) and CLI specifications. Thus, the IL code insinuates the following processing for the evaluation of \( \text{exp}_1.f \) in order to produce the variable: \( \text{exp}_1 \) is evaluated and the meaning of \( f \) in the context of the type of \( \text{exp}_1 \) is determined.

**Fig. 2.** Illegal JIT optimization: under .NET, a DivideByZeroException is thrown instead of a NullReferenceException.
Fig. 3. The ASM rules for the execution of a field assignment.

\[
\begin{align*}
\text{exp}_1.t::f &= \text{exp}_2 \rightarrow \text{pos} := \text{exp}_1 \\
\text{val}_1.t::f &= \text{exp}_2 \rightarrow \text{pos} := \text{exp}_2 \\
\text{val}_1.t::f &= \text{val}_2 \rightarrow \text{if} \ \text{val}_1 = \text{null} \ \text{then} \ \text{FAILUP}(\text{NullReferenceException}) \\
\text{else} \ \text{SETFIELD}(\text{val}_1, t::f, \text{val}_2) \\
&\quad \text{YIELDUP}(\text{val}_2)
\end{align*}
\]

(through a lookup). Note that there is no null check yet. The null check for the left-hand side of the assignment is only at the end (IL_000f), namely just before the result of the addition is stored in x ([3, Partition III, §4.26]). This is reflected in the C# model by the rules (refined for C#e) from Fig. 3.

The example Test9 seems to support this timing for the null test since, under the same conditions, \(a.x = 1/i\) throws a \text{DivisionByZeroException}.

The next step after producing the variable for \(a.x\) is the evaluation of the right-hand side of the assignment (i.e. \(a.x + 1/i\)), where \(a.x\) has to be evaluated first: \textit{operands in an expression are evaluated from left to right} ([2, §7.2]). This order is certified also by the IL code from Fig. 2. This time, the value of the variable represented by \(a.x\) is needed and not the variable. There is no doubt that, at least now, a \text{NullReferenceException} should be thrown. This also agrees with the description of \text{ldfld} in [3, Partition III, §4.9].

So, despite the unclear formulation provided by the specifications, with both approaches, a \text{NullReferenceException} has to be thrown. Surprisingly, under .NET we obtain a \text{DivideByZeroException}.

**A Possible Explanation.** We get that this strange behavior because it seems that \(1/i\) was evaluated first. This can be observed from the output of the JIT compiler where, probably due to a too aggressive optimization, the division by zero is executed before the two kinds of evaluations for \(a.x\), that otherwise would have thrown a \text{NullReferenceException}. Note that the order of evaluation cannot be changed in our case since \textit{throwing of exceptions} are considered side-effects whose order has to be preserved as stated in [2, §3.10]. Therefore, this optimization performed by the JIT of .NET is illegal. Under Rotor and Mono, the evaluation of these assignments is done in the appropriate order and throws a \text{NullReferenceException}. If we assume that both kinds of exceptions are handled in some catch blocks, we would easily get different results for the same small program run under .NET and Rotor.

Although, due to optimizations, the .NET C# compiler could be faster than the Java compiler on some particular benchmarks, one has to care at such illegal optimizations. We may figure out that Rotor behaves correctly in this situation because of the naive implementation of its JIT compiler.
4 Implementation Problems

The present section brings to light some mistakes in different C# implementations. In each case, we explicitly display the discovered problem and we indicate what could be the reason for it.

4.1 The Delegates Invocation List

The Problem. A feature of C# that has no correspondent in Java is the notion of function pointers. This is addressed in C# by means of delegates. There are various ways of creating a delegate instance. The one we will refer to here is the one that uses another delegate instance of the same delegate type. The reference manual [2] states in §7.5.10.3 that the newly created delegate refers to the same invocation list as the other delegate. The example Test10 shows that the two delegates do not have the same invocation list. Moreover, in general the lists do not even have the same length. Thus, let D be the following delegate type:

```csharp
delegate void D(string s);
```

We create in the `Main` method two delegate instances of type D: an instance `x` that references the method `Write` with an instance of type `TextWriter` (returned by the standard output stream property `Out` declared in class `Console`) and an instance `y`:

```csharp
D x = new D(Console.Out.Write);
D y = new D(x);
```

We now intend, to obtain the invocation lists of `x` and `y` in the `Main` method. In our case, this can be done using the readonly properties `Method` and `Target` declared in class `System.Delegate` as described in Test10. Thus, the outputs under .NET and Rotor report for `x` the method `Void Write(System.String)` and the target object `System.IO.TextWriter+SyncTextWriter`, while for `y` we have the method `Void Invoke(System.String)` and target object `null`. The test `x == y` has a negative result. The example shows that the invocation list of the newly created delegate is a singleton list with the element `(null,D::Invoke)`. Note that `D::Invoke` is an instance method, and the target object on which was supposed to be invoked is `null` (value required for a static method).

A Possible Explanation. It seems that .NET makes an exception and it does not treat this case as the manual states, namely invoking `D::Invoke` on `null`. What actually happens is that whenever the delegate `y` is invoked, `D::Invoke` is invoked on `x`. Thus, we have a double redirection since anyway, in the step, the methods in the invocation list of `x` are going to be invoked. Note that, with respect to `D::Invoke`, `mono` and `mint` provide the correct target object for `y`.

The discrepancy showed up when we tried to figure out the ASM rule for such a kind of delegate creations. The rule shows the incoherence between the specifications and the .NET implementation (but still fixing the target object):
new D(\textit{ref}) \rightarrow \\
\textbf{if} ref = \textit{null} \textbf{then} FAILUP(NullReferenceException) \\
\textbf{else} let d = new(Ref, D) \textbf{in} \\
\textbf{runTimeType}(d) := D \\
// C# Specification §7.5.10.3 \\
\textit{invocationList}(d) := \textit{invocationList}(\textit{ref}) \\
// .NET \\
// \textit{invocationList}(d) := [(\textit{ref}, D::\textit{Invoke}(S_1, \ldots, S_n))] \\
\textbf{YIELDUP}(d)

### 4.2 The Delegates Creation

**The Problem.** The next aspect about the delegates creations brings again into the picture the exact time of initializing a class discussed in Section 2.1. Let us suppose we have a delegate creation \texttt{d = new D(A.m)} like in example Test11, where \texttt{m} is a static method declared in the class \texttt{A}, which declares explicitly its static constructor.

Regarding the static method \texttt{m} in \texttt{d = new D(A.m)}, [2, §15.2] states doubtless that it is referenced. This must trigger the initialization of \texttt{A} ([2, §10.11]) since the class is not marked with \texttt{beforfieldinit}. If we run \textbf{Test11} under .NET, Rotor or under Mono with the \textit{mint} interpreter, it yields an output that indicates the inverse order: the static constructor of \texttt{A} is executed after the creation of \texttt{d} and before \texttt{d} is invoked. Under Mono, \textit{mono} treats this case correctly (according to the specifications) and first initializes class \texttt{A}.

**A Possible Explanation.** The reason for the above behavior could be a mistake in the reference manual otherwise this is clearly a bug. The doubt is concerning the qualification referenced. Shouldn’t we find in the Specification accessed instead of referenced, i.e. the static constructor of a class executes before any of the static members (excepts constants) declared by the class are accessed? Actually, there is no reason to require the initialization of \texttt{A} before the delegate creations: the compiler simply retrieves the token for \texttt{A::m} from the meta data, and it substitutes it in the constructor call for \texttt{d}.

In order to certify the ambiguity for referenced, note also the example from Fig. 4 (Test12), where Rotor and Mono (mint) do not consider the correct moment for the initialization of the class \texttt{A} when its static field \texttt{A.x} is assigned. Note that, since \texttt{A} is not a \texttt{beforfieldinit} type, independent of the implementation, it should be clear when exactly gets class \texttt{A} initialized.

In this case, \texttt{A.x} is clearly referenced and the attempt of getting the variable should trigger the initialization of class \texttt{A}. If this happens, then, no matter when exactly (before \texttt{A}) \texttt{B} was initialized, the field \texttt{B.y} would be 2. Surprisingly, if we run the program in Rotor or \textit{mint} runtimes, the field \texttt{B.y} has still the value 1, which means that class \texttt{A} has not been initialized. If the same program is run under .NET, we get for \texttt{B.y} the correct value 2. This means that \texttt{A} is initialized before the division by zero and therefore, when the exception is caught, the field \texttt{B.y} has the value 2. For this example, the \textit{mono} runtime encounters a problem and closes.
class B {
    public static int y = 1;
}
class A {
    public static int x;
    static A() {
        B.y = 2;
    }
}
class Test {
    public static void Main(String[] args) {
        int i = 0;
        try {
            A.x = 1 / i;
        } catch (DivideByZeroException e) {
            Console.WriteLine("B.y = {0}",B.y);
        }
    }
}

Fig. 4. Under Rotor and mint, A is not initialized before the assignment to A.x.

4.3 The Definite Assignment Analysis

The Problem. The following case (Test13) is a simplification of the example presented in the specifications [2, §5.3.3.15]: we have a declaration of the local variable i and the try-finally statement followed by a labeled statement

```csharp
try { goto lab; }
finally { i = 3; }
lab: Console.WriteLine("i = {0}" , i);
```

All the execution paths leading to this printing should pass through the finally block, which initializes i. Thus, according to the specifications [2, §5.3.3.1], i should be definitely assigned before the labeled statement. But if we compile this example under .NET Framework 1.0, Rotor or Mono, we get the error that i is unassigned.

A Possible Explanation. The error could arise from the definite assignment analysis performed by a C# compiler. The corresponding details concerning this analysis go much beyond this paper (see [16]). In the meantime, this error was fixed in .NET Framework 1.1 but still exists in Rotor.

4.4 The Constraints for an Array Element

The Problem. Another incoherence between the specifications and the .NET and Rotor implementations of C# refers to the constraints for an array element.
The reference manual [2, 14.5.6.1] imposes for an array element \( e[I_1, I_2, \ldots, I_n] \) the following condition for the types of the indices: each \( I_x \) is an expression of type \text{int}, \text{uint}, \text{long}, \text{ulong}, \text{or of a type that can be implicitly converted to one or more of these types}.\] Let us consider a rectangular array \( a \) and two local variables \( i \) and \( j \) (like in \text{Test14}):

```csharp
int[,] a = new int[2, 2];
ulong i = 1;
long j = 1;
```

Satisfying the stated condition, an access to \( a[i,j] \) should not cause any troubles. But under .NET and Rotor, we get the following compile-time errors:

```
error CS0029: Cannot implicitly convert type 'ulong' to 'int'
error CS0029: Cannot implicitly convert type 'long' to 'int'
```

The message is quite confusing: why would it try to convert \text{ulong} and \text{long} to \text{int}, instead of (implicitly) converting \text{ulong} and \text{long} to \text{ulong}, respectively \text{long} (as the specifications indicate)? Under Mono, the mcs compiler does not complain.

**A Possible Explanation.** The example seems somehow to sustain that the following constraint is more plausibly the one which is implemented: **all \( I_x \) can be implicitly convertible either to \text{long} or to \text{ulong}.** Therefore this is rather a mistake in the specifications than a bug in the implementation.

### 5 Some Other Remarks

This section contains some comments concerning certain unnatural implemented issues and one simple mistake within an example from the specifications.

#### 5.1 The Best Method

The following aspect is about how the best method to invoke is determined in case of a method invocation. Let us suppose we have a method invocation. In order to determine the best method for the run-time processing during compile-time within the method lookup, all the \text{override} methods are removed from the set of candidate methods [2, §7.3]. As a consequence, it may happen that an \text{override} method, which is better than another candidate method (with respect to the set of argument types), is removed from the candidates. The example \text{Test15} provides such a situation. It considers the class \( A \) which defines the method \( m \):

```csharp
public virtual void m(int i) {...}
```

and a subclass \( B \) of \( A \) which has two overloaded methods \( m \):

```csharp
public override void m(int i) {...}
public void m(long i) {...}
```
The invocation expression b.m(i) from the Main method, where b is an instance of type B and i is a local variable of type int, yields under .NET and Rotor an output where B::m(long) is invoked. This is, however, according to the specifications since the override method B::m(int) is not a candidate even if it is better than B::m(long) and since B::m(long) is applicable, all the methods declared in base classes of B (in particular A::m(int)) are removed from the candidates. So, although B::m(int) specializes an existing inherited virtual method by providing a new implementation, it seems that in .NET and Rotor, this specialization is not used in this case. It might be that this is a mistake in the specifications and .NET and Rotor implement it exactly. Under Mono, both mono and mint invoke the better method B::m(int). It is an example that many programmers could easily get wrong.

5.2 Delegates with Empty Invocation List
Something else that can be considered surprising is the design of the delegates which is inappropriate for the case of a delegate instance with an empty invocation list. If we follow the specifications, one can get such a delegate instance only through a delegate removal ([2, §7.7.5]). It is stated that, if the removal of the second delegate invocation list from the first delegate invocation list results in an empty list, then the resulted delegate is null. This proves that a delegate d of type D can have an empty invocation list if and only if d is null. Therefore if d is invoked, a NullPointerException would be thrown because the method D::Invoke is called on d. It would seem more natural if this kind of invocation would simply return (i.e. nothing happens) since there is no method in the list to invoke.

5.3 The Type of Array Type Variables
The next remark is about the array types. Let a be an array type variable:

```csharp
int[,] a = new int[2,3][];
```

If we want to determine the type of a (like in example Test16) using the instance method GetType defined by the class object:

```csharp
Console.WriteLine(a.GetType());
```

we will get under all C# implementations System.Int32[,][,] which is not exactly what we expect: System.Int32[,][]. It seems that in the output, the arrays are read from right to left.

5.4 A Mistake
The last remark of this section shows a simple mistake in the specifications of the indexers. In [2, §10.8], there is a C# program (Test17) which implements the sieve algorithm to compute the number of primes between 1 and a given number. If we run this program to count the number of primes between 1 and 2 we get 2, between 1 and 3, we get 3. The reason behind this mistake is simple: the counting should start with 0 (count = 0) instead of 1 (count = 1).
Acknowledgment

We gratefully acknowledge partial support of the overall work [1] by a Microsoft grant within the ROTOR project during the year 2002-2003. We would like to thank also to the anonymous referees for their valuable remarks on the submitted version of this paper.

References

Abstract. Static single assignment (SSA) form is the intermediate representation of choice in modern optimizing compilers for which no formal semantics has been stated yet. To prove such compilers correct, a formal semantics of SSA representations is necessary. In this paper, we show that abstract state machines (ASMs) are able to capture the imperative as well as the data flow-driven and therefore non-deterministic aspects of SSA representations in a simple and elegant way. Furthermore, we demonstrate that correctness of code generation can be verified based on this ASM semantics by proving the correctness of a simple code generation algorithm.

1 Introduction

Because static single assignment (SSA) representations allow for the explicit representation of data flow as well as control flow dependencies, they are the preferred intermediate representation in modern optimizing compilers. Optimizations in compilers are typically the most error-prone parts, cf. e.g. [New01]. Such errors can only be eliminated if the applied optimizations are verified. To prove them correct, we first of all need a formal semantics of the employed intermediate representation. In this paper, we state a formal semantics for SSA representations. We require it to capture the imperative nature of SSA representations as well as their non-deterministic data flow driven character equally well. Furthermore, it should be applicable in correctness proofs for optimizing compilers.

Our semantics for SSA representations is formalized as an abstract state machine (ASM) [Gur95]. Each state during computation characterizes the current basic block. We capture the imperative, state-based part of SSA computations by transition rules which transfer control flow from one basic block to its successor basic block. Within basic blocks, SSA computations are purely data flow driven. We model these computations by transition rules which are non-deterministic in the sense that at a given point during the run of the ASM, more than one rule might be applicable. Our specification of SSA semantics is well-suited to prove the correctness of code generation algorithms. In this paper, we prove the correctness of a relatively simple machine code generation algorithm. Thereby we prove that a generated deterministic machine program preserves the data flow
dependencies of the source SSA program by showing that the sequence of applied transition rules during the execution of the machine program corresponds to one possible sequence of transition rules in the non-deterministic SSA semantics. Furthermore, we point out how this proof can be extended to capture also more complex optimization strategies during code generation.

This paper is structured as follows: First we introduce SSA representations in section 2. Then we describe how ASMs are typically used in the specification of the formal semantics of programming languages, cf. section 3. Afterwards, in section 4, we state our formal semantics for SSA representations based on ASMs. In section 5, we demonstrate that this formal semantics is a well-suited basis for correctness proofs of optimizing compilers. We complete this paper with a discussion of related work in section 6 and the conclusions in section 7.

2 SSA Intermediate Representations

Static single assignment (SSA) form has become the preferred intermediate program representation for handling all kinds of program analyses and optimizing program transformations prior to code generation [CFR+91]. Its main merits comprise the explicit representation of def-use-chains and, based on them, the ease by which further dataflow information can be derived.

By definition SSA-form requires that a program and in particular each basic block\(^1\) is represented as a directed graph of elementary operations (jump/branch, memory read/write, arithmetic operations on data) such that each ”variable” is assigned exactly once in the program text. Only references to such variables may appear as operands in operations. Thus, an operand explicitly indicates the data dependency to its point of origin. The directed graph of an SSA-representation is an overlay of the control flow and the data flow graph of the program. A control node may depend on a value which forces control to conditionally follow a selected path. Each basic block has one or more such control nodes as its predecessor. At entry to a basic block, \(\phi\) nodes, \(x = \phi(x_1, \ldots, x_n)\), represent the unique value assigned to variable \(x\). This value is a selection among the values \(x_1, \ldots, x_n\) where \(x_i\) represents the value of \(x\) defined

---

\(^1\) A program is divided into basic blocks by determining maximal sequences of instructions that can be entered only at their first and exited from their last instruction.
on the control path through the \( i \)-th predecessor of the basic block. \( n \) is the number of predecessors of the basic block. Programs can easily be transformed into SSA representation, cf. [Muc97], e.g. during a tree walk through the attributed syntax tree. The standard transformation algorithm subscripts each variable. At join points, \( \phi \) nodes sort out multiple assignments to a variable which correspond to different control flows through the program.

As example, figure 1 shows the SSA representation for the program fragment:

\[
a := a+2; \text{ if(...) } \{ a := a+2; \} \ b := a+2;
\]

In the first basic block, the constant 2 is added to \( a \). Then the \textit{cond} node passes control flow to the ‘then’ or to the ‘next’ \textit{block}, depending on the result of the comparison. In the ‘then’ \textit{block}, the constant 2 is added to the result of the previous \textit{add} node. In the ‘next’ \textit{block}, the \( \phi \) node chooses which reachable definition of variable ‘a’ to use, the one before the if statement or the one of the ‘then’ \textit{block}. The names of variables do not appear in the SSA form. Since each variable is assigned statically only once, variables are identified with their value.

SSA representations describe imperative, i.e. state-based computations. A virtual machine for SSA representations starts execution with the first basic block of a given program. After execution of the current basic block, control flow is transferred to the uniquely defined subsequent basic block. Hence, the current state is characterized by the current basic block and by the outcomes of the operations contained in the previously executed basic blocks.

Memory accesses need special treatment. In the functional store approach [Ste95], memory read/write nodes may be considered as accesses to fields of a global state variable \textit{memory}. A memory write access modifies this global variable \textit{memory} and requires that the outcome of this write operation yields a new (subscribed) version of the \textit{memory} variable. These duplications of the \textit{memory} variable are the reason for inefficiencies in practical data flow analyses. As a solution, one might try to determine which memory accesses address overlapping memory areas and thus are truly dependent on each other and which address independent parts with no data dependencies. For the purpose pursued in this paper, these considerations are irrelevant. It is our goal to define a formal semantics for SSA representations. The same semantic description can be used for accesses to only a single as well as for accesses to several independent memories.

3 Abstract State Machines (ASMs)

ASMs [Gur95] are a general computation model to describe all kinds of computations as e.g. programming languages, hardware architectures, distributed systems, or real-time protocols. Especially for programming languages, many specifications of existing languages exist (e.g. C [GH93], C++ [Wal95], Java [SSB01], and SDL [EGGP00]). In this section, we summarize ASMs with respect to this particular use in the specification of programming languages.

3.1 Semantics of Programming Languages

The semantics of programming languages is in general compositional. Given a program in form of its abstract syntax tree, the semantics of each node can be
defined directly given its immediate successors. Nevertheless, certain constructs in programming languages exhibit a semantics which is inherently not compositional. E.g., goto-statements may leave a program part and go to some other place which cannot be described via the predecessor or successor relation in abstract syntax trees. ASM semantics is able to describe such non-compositional semantics. Therefore, continuations are defined. They describe where to proceed with the computation. If the control flow branches at a node, then several such continuations are defined, each describing the succeeding computation depending on the branch direction. The abstract syntax tree together with these continuations is the basis to define the semantics of programming languages.

Hence, ASMs are a well-suited framework to define a formal SSA semantics, cf. section 4, because they can utilize the explicitly stated continuations directly.

### 3.2 ASM Semantics for Programming Languages

Abstract state machines describe the semantics of programming languages operationally as state transition systems based on the abstract syntax trees of programs. Part of the current state is the current task, a pointer to the node in the abstract syntax tree currently executed. During program execution, states are transformed into new states, thereby also updating the pointer to the current task. States are regarded as algebras over a given signature. Each n-ary function symbol is interpreted with an n-ary mapping. During a state transition, the interpretation I of some of the function symbols may change. E.g., if a function symbol S specifies the state of memory, then a variable assignment x:=v changes the interpretation I(S) of the function symbol S for argument x: I(S(x)) := I(v) holds in the new state. In general, an ASM consists of four components (Σ ∪ Δ, A, Init, Trans): The signature is composed of two disjoint sorted signatures, the signature of the static functions Σ and the signature of the dynamic functions Δ. A is the static algebra, an order-sorted Σ-algebra interpreting the function symbols in Σ. Init is a set of equations over A defining the initial states of A. Trans is a set of transition rules for specifying the state transitions by defining or updating, resp., the interpretations of certain function values of functions in Δ. A (Σ ∪ Δ)-algebra is a state of the ASM iff its restriction to Σ is the static algebra A. If q is a state, f ∈ Δ is a function symbol, and t_i are terms over Σ ∪ Δ with interpretations x_i in q, then update f(t_1, ..., t_n) := t_0 defines the new interpretation of f in the succeeding state q’ as

\[
q’ \models f(x_1, \ldots, x_n) = \begin{cases} 
  x_0 & \text{if for all } i, 0 \leq i \leq n, \ q \models t_i = x_i \\
  f_q(x_1, \ldots, x_n) & \text{otherwise}
\end{cases}
\]

A transition rule defines a set of updates which are executed in parallel:

\[
\text{if } \text{Cond} \ \text{then } \text{Update}_1 \ldots \text{Update}_n \ \text{fi}
\]

If \( q \models \text{Cond} = \text{true} \) in state q, then Update_1 \ldots Update_n are executed in q.
When defining the semantics of programming languages, the abstract syntax tree is used as basis and meaning is attached to it. Thereby, it is assumed that the abstract syntax tree contains attributes defining all continuations, especially for the non-compositional changes of the control flow. The definition of the ASM models the program counter during program execution, thereby using the continuation attributes which might be split up according to the value of conditions (true case and false case). Here is the example of a transition rule defining the semantics of the while-loop, as stated in [GZ99]. $CT \ (CT = \text{current task})$ is the abstract program counter, $CT.TT$ (true task) is the true-continuation attribute of $CT$ and $CT.FT$ (false task) is the false-continuation attribute of $CT$.

\[
\begin{align*}
\text{if } & CT \in \text{While} \text{ then} \\
& \quad \text{if } \text{value}(CT.\text{cond}) = \text{true} \text{ then } CT := CT.TT \\
& \quad \quad \text{else } CT := CT.FT \text{ fi fi}
\end{align*}
\]

The semantics of each program node is described by a finite set of transition rules. Typically the condition of such a transition rule specifies the nodes in the abstract syntax tree (\textbf{While}-nodes in our example) for which the transition rule is applicable. The transition rules define updates, thereby employing children nodes (in our example $CT.\text{cond}$) and statically computed continuations ($CT.TT$ and $CT.FT$ in our above example). In the remainder of this paper, we show how the semantics of SSA representations can be specified with ASMs. Furthermore, we prove the correctness of code generation based on this specification.

## 4 An ASM Semantics for SSA Representations

In this section, we present an ASM semantics for SSA representations. We first show how the structure of SSA programs is defined. Then we proceed by defining the transition rules describing the dynamic behavior of SSA programs.

\[
\begin{array}{|l|}
\hline
\text{belongs\_to} & : \text{Operations} \rightarrow \text{BasicBlocks} \\
\text{Pred}_1, \text{Pred}_2, \text{Pred}_3 & : \text{Operations} \rightarrow \text{Operations} \\
\text{Preds} & : \text{Operations} \rightarrow \text{List}(\text{Operations} \times \text{BasicBlocks}) \\
\text{select} & : \text{List}(\text{Operations} \times \text{BasicBlocks}) \times \text{BasicBlocks} \rightarrow \text{Operations} \\
\text{kind} & : \text{Operation} \rightarrow \{\phi, \text{add}, \text{AND}, \text{read}, \text{write}, \text{jump}, \text{branch}\} \\
\text{initial\_store} & : \mathbb{N} \rightarrow \{\text{undef}\} \text{ s. t. } \forall n \in \mathbb{N}. \text{initial\_store}(n) = \text{undef} \\
\hline
\end{array}
\]

\textbf{Fig. 2. Static Functions}

Programs in SSA form are characterized by their basic blocks and by the connection between them. Each basic block contains \(\phi\) operations, arithmetic and Boolean operations\(^2\) as well as memory accesses. Moreover, a jump or branch operation is contained which transfers control flow during computation to the succeeding basic block. Note that each operation belongs uniquely to one specific

\(^2\) For sake of simplicity, we only consider the “add” and “AND” operation as representatives of arithmetic and Boolean operations.
basic block. During computation, the SSA representation itself is not modified. Hence, we describe the static structure of SSA programs by the static functions of the ASM. They are listed in figure 2.

The function \textit{belongs_to} specifies, for each operation \( op \in \text{Operations} \), to which basic block \( b \in \text{BasicBlocks} \) it belongs. SSA representations specify explicitly the data flow of a program. In our formalization, this is expressed with the functions \( \text{Pred}_1, \text{Pred}_2, \) and \( \text{Pred}_3 \). \( \text{Pred}_i(op) = op' \) means that the result of \( op' \) is the \( i \)th input of operation \( op \), i.e., \( op' \) is the \( i \)th predecessor of \( op \). For the evaluation of \( \phi \) nodes, we need to know what their predecessors are and to which basic blocks they belong. We represent this information by a list of pairs \( \text{Preds} \). Each such pair consists of a predecessor operation and the basic block to which this predecessor operation belongs. The function \textit{select} returns, for a given basic block \( b \), the operation \( op \) such that \((op, b)\) is an element of the list \( \text{Preds} \). \textit{kind} is a function which returns for each operation its name. We specify memory accesses according to the functional store approach [Ste95]. In particular, we model the store as a function which maps the natural numbers, i.e. the potentially infinitely many store cells, into the set of possible values, i.e. \((\text{Bool} \cup \mathbb{Z} \cup \{\text{def}, \text{undef}\})\). Initially, the content of each cell is \text{undef}, represented by the constant static function \textit{initial_store}. When executing memory write operations, this assignment of values might be changed. Hence, in general, arbitrary functions mapping from \( \mathbb{N} \) into \((\text{Bool} \cup \mathbb{Z} \cup \{\text{def}, \text{undef}\})\) represent the value of memory during computation. In the functional store approach, the result of a memory write is the updated memory. Hence, the dynamic function \textit{value} which assigns operations to their results maps also into this set of functions from \( \mathbb{N} \) into \((\text{Bool} \cup \mathbb{Z} \cup \{\text{def}, \text{undef}\})\), cf. figure 3.

\begin{verbatim}
value : Operations -> Value \cup \{def, undef\} \cup (\text{Bool} \cup \mathbb{Z} \cup \{\text{def}, \text{undef}\})^\mathbb{N}
init : Bool
current_block, next_block, pred_block : BasicBlocks
\end{verbatim}

\textbf{Fig. 3. Dynamic Functions}

Basic blocks are evaluated by first evaluating the \( \phi \) nodes, by then computing the arithmetic, Boolean and memory operations, and finally by computing the successor basic block. The dynamic constant \textit{init} guides the evaluation of the \( \phi \) nodes. During their evaluation, \textit{init} is \text{true}, afterwards it is set to \text{false}. A state during computation is characterized by the current basic block \textit{current_block}, by its predecessor block \textit{pred_block}, and by its current evaluation phase represented by the value of \textit{init}. Figure 3 summarizes the dynamic functions.

\textbf{Remark:} Note that the \( \phi \) nodes of a block must be evaluated before its other operations. Otherwise, in case that a block is its own predecessor, the evaluation of a succeeding operation might falsify the result of a delayed \( \phi \) node evaluation.

Initially, the current block is \textit{start_block}, a distinguished basic block representing the starting point of computation. Furthermore, there is another
distinguished basic block \texttt{start\_pred} representing the predecessor block of \texttt{start\_block}. Since a program might have input values, we need to represent them in the SSA form. We do this with the block \texttt{start\_pred} which contains constants representing the program inputs. The initial value for the successor block is \texttt{void} since no successor block has been computed yet. Also, no operation has been computed, hence all their values are set to \texttt{undef}. The dynamic constant function \texttt{init} is set to \texttt{true} initially because computation starts with the evaluation of the \texttt{phi} nodes.

Evaluation of a basic block starts with the evaluation of the \texttt{phi} nodes, cf. figure 5. All \texttt{phi} nodes are evaluated simultaneously, \texttt{init} is set to \texttt{false}, and the values of all other operations in the current basic block are reset to \texttt{undef}.

\begin{verbatim}
if init and current_block \neq void then
  forall op \in \{op' | belongs_to(op') = current_block \land kind(op') = \phi\}
    do in parallel value(op) = select(Preds(op), Pred(op)) od;
  forall op \in \{op' | belongs_to(op') = current_block \land kind(op') \neq \phi\}
    do in parallel value(op) = undef od;
  init := false fi
\end{verbatim}

**Fig. 5.** Transition Rule for the Evaluation of \texttt{phi} nodes

After evaluating the \texttt{phi} nodes, \texttt{init} is \texttt{false}, and rules for arithmetic, Boolean, memory access as well as jump and branch operations might be applied whenever their conditions are fulfilled, i.e., whenever their input values are computed. In general, there might be several operations within one basic block which can be evaluated. This stems from the data-flow driven nature of computations within SSA basic blocks. The selection among the corresponding updates is non-deterministic. We formulate these non-deterministic updates with the choose constructor [Gur95]. In section 5 we show that this non-deterministic definition is sound in the sense that each evaluation order which fits to the acyclic data dependencies of SSA basic blocks yields the same result. Note that our notation “\texttt{choose Op} \subseteq \{op | some requirements \} \texttt{ in } \mathcal{P}(\text{Operations}) R_0 \texttt{ endchoose}” in figure 6 ($\mathcal{P}(S)$ denotes the powerset of a given set $S$) is an abbreviation for the qualified choose construct: “\texttt{choose v in U satisfying g(v) R_0 endchoose}”.

The rules for arithmetic and Boolean operations as well as for memory accesses follow the same schema: If the operation belongs to the current block, if its predecessor operations are evaluated, if the \texttt{phi} nodes of the current block are evaluated (\texttt{not init}), and if the operation itself has not yet been evaluated, then the operation can be evaluated. Memory access operations are specified according to the functional store approach [Ste95]. Thereby, memory is itself treated as a global variable. Each write operation modifies the store and, hence, the value

\begin{verbatim}
current\_block = start\_block
pred\_block = start\_pred
next\_block = void
\forall op \notin \text{start\_pred.value(op)} = \text{undef}
init = \text{true}
\end{verbatim}

**Fig. 4.** Initial States
choose \( \text{Op} \subseteq \{ \text{op} \mid \text{kind}(\text{op}) \in \{ \text{add, AND, read, write, jump, branch} \} \) and
\[\begin{align*}
\text{belongs_to}(\text{op}) &= \text{current_block} \quad \text{and} \quad \text{value}(\text{op}) = \text{undef} \quad \text{and} \\
\text{value}(\text{Pred}_1(\text{op})) &\neq \text{undef} \quad \text{and} \quad (\text{kind}(\text{op}) \in \{ \text{add, AND, read, write, branch} \} \\
&\Rightarrow \text{value}(\text{Pred}_2(\text{op})) \neq \text{undef}) \quad \text{and} \\
(kind(\text{op}) \in \{ \text{write, branch} \} \Rightarrow \text{value}(\text{Pred}_3(\text{op})) \neq \text{undef}) \} \in \mathcal{P}(\text{Operations})
\end{align*}\]

\[
\begin{align*}
\text{if } \text{op} \in \text{Op} \text{ and } \text{kind}(\text{op}) = \text{add} \text{ and } \neg \text{init} \text{ then} \\
\text{value}(\text{op}) := \text{value}(\text{Pred}_1(\text{op})) + \text{value}(\text{Pred}_2(\text{op})) & \text{ fi} \\
\text{if } \text{op} \in \text{Op} \text{ and } \text{kind}(\text{op}) = \text{AND} \text{ and } \neg \text{init} \text{ then} \\
\text{value}(\text{op}) := \text{value}(\text{Pred}_1(\text{op})) \wedge \text{value}(\text{Pred}_2(\text{op})) & \text{ fi} \\
\text{if } \text{op} \in \text{Op} \text{ and } \text{kind}(\text{op}) = \text{read} \text{ and } \neg \text{init} \text{ then} \\
\text{value}(\text{op}) := \text{value}(\text{Pred}_1(\text{op}))(\text{value}(\text{Pred}_2(\text{op}))) & \text{ fi} \\
\text{if } \text{op} \in \text{Op} \text{ and } \text{kind}(\text{op}) = \text{write} \text{ and } \neg \text{init} \text{ then} \\
\text{value}(\text{op}) := \text{value}(\text{Pred}_1(\text{op}))\text{value}(\text{Pred}_1(\text{op}))(\text{value}(\text{Pred}_2(\text{op}))) := \text{value}(\text{Pred}_3(\text{op})) & \text{ fi} \\
\text{if } \text{kind}(\text{op}) = \text{jump} \text{ and } \neg \text{init} \text{ then} \\
\text{next_block} := \text{Pred}_1(\text{Op}); \text{ value}(\text{op}) := \text{def} & \text{ fi} \\
\text{if } \text{kind}(\text{op}) = \text{branch} \text{ and } \neg \text{init} \text{ then} \\
\text{if } \text{value}(\text{Pred}_1(\text{op})) \text{ then } \text{next_block} := \text{Pred}_2(\text{op}) \text{ else } \text{next_block} := \text{Pred}_3(\text{op}) & \text{ fi; value}(\text{op}) := \text{def} \text{ fi}
\end{align*}
\end{equation}

Fig. 6. Transition Rule for Arithmetic, Boolean, Memory, Jump, Branch Operations

of this global variable. In SSA representations, modifications of variable values lead to duplications of that variable, i.e. to different copies, cf. also section 2. In case of the memory write operation, the result is a “new” memory which is identical with the old one except for the point which has been written by the write operation. In figure 6, the transitions for read and write operations are given. The read operation has two predecessor operations. The first gives the memory, the second the address to be read. The write operation has an additional predecessor, the value to be written to the address indicated by the second predecessor. Result of the read operation is the value stored in memory at the indicated address, result of the write operation is the updated memory.

\[
\begin{align*}
\text{if } \{ \text{op} \mid \text{belongs_to}(\text{op}) = \text{current_block} \text{ and } \text{value}(\text{op}) = \text{undef} \} = \emptyset \text{ then} \\
\text{pred_block} := \text{current_block}; \text{ current_block} := \text{next_block}; \text{ init} := \text{true} & \text{ fi}
\end{align*}
\]

Fig. 7. Transition Rule for Block Transition

Jump and branch operations can also be evaluated if they belong to the current block and if the evaluation of the \( \phi \) nodes has been completed (\( \neg \text{init} \)), even if there are other unevaluated operations (arithmetic, Boolean, memory read/write) in the basic block. The corresponding transition rules are also stated in figure 6. They set the value of the dynamic constant \( \text{next_block} \). The transition to the succeeding basic block itself takes place by executing another transition
rule, namely the one specified in figure 7. Its condition states that all operations must have been evaluated before transition to the next block can be done.

In summary, semantics of SSA representations can be described directly with ASMs. The imperative, state-based part is captured by transition rules which transfer control flow from one basic block to its successor block and by the distinction between the initial phase which evaluates the predecessor nodes of the current block and the succeeding initial phase which computes the remaining operations. This evaluation of the remaining operations is purely data-driven. It is captured very elegantly by non-deterministic transition rules such that at a given point during computation, more than one possible set of updates might be applicable.

5 Proof of Correctness for Code Generation in Compilers

Code generation in compilers transforms the intermediate representation into a sequence of machine instructions. When using SSA as intermediate form, one has to cope with imperative control flow between basic blocks as well as with the purely data-driven evaluation of operations within basic blocks. In contrast, the machine language is purely imperative because one instruction is executed after the other. To ensure correctness of machine code generation, one needs to prove that the semantics of the SSA representation is the same as the semantics of the generated machine code. For this proof, we need a formal semantics of the target machine language which we specify with ASMs in subsection 5.1. Typically, code generation is divided into code selection, instruction scheduling, and register allocation. To be able to concentrate on the essentials of the correctness proof, we consider a relatively simple code generation algorithm in subsection 5.2, prove its correctness in subsection 5.3, and discuss possible extensions in 5.4.

5.1 ASM Semantics for the Machine Language

We consider the machine language with the instructions summarized in table 1. Each instruction $i$ may have a label $l$: “$l : i$”. For simplicity, we assume that the machine can use arbitrarily many registers. The semantics of this machine language is formally specified with the ASM in figure 8. The static functions $NI$ (next instruction), $LI$ (labelled instruction), $TI$ (true instruction), and $FI$ (false instruction) are used to specify the order of instructions in the machine program. In the sequential case, one instruction is executed after the other. The function $NI$ maps each instruction to its successor instruction. In case of JMP operations, the succeeding instruction is the one at the designated label. The function $LI$ maps each JMP operation to this successor instruction. In case of a BRN operation, we distinguish between the label in the positive and negative case, specified with the functions $TI$ and $FI$. $TI$ ($FI$) maps the current instruction to the instruction at label $l$ (12). The dynamic functions $reg\_table$ and $Memory$ map registers and memory addresses to their current content which is initially $undef$. The transition rules in figure 8 specify the dynamic semantics of the machine language. Thereby, the dynamic constant $curr\_instr$ serves as a pointer to the current instruction which is to be executed. In the initial states, it is initialized with the first instruction of the machine program.
Table 1. Machine Language

<table>
<thead>
<tr>
<th>Operation</th>
<th>Informal Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD R1, R2, R3</td>
<td>adds register contents of R1, R2 and writes result in R3</td>
</tr>
<tr>
<td>AND R1, R2, R3</td>
<td>computes conjunction of contents of R1, R2 and writes result in R3</td>
</tr>
<tr>
<td>READ R1,R2</td>
<td>writes memory content at address stored in R1 into R2</td>
</tr>
<tr>
<td>WRITE R1, R2</td>
<td>writes value of R2 at address stored in R1</td>
</tr>
<tr>
<td>CP R1, R2</td>
<td>copies value of R1 into R2</td>
</tr>
<tr>
<td>JMP l</td>
<td>jumps to instruction with label l</td>
</tr>
<tr>
<td>BRN R1, l1, l2</td>
<td>branches to instruction with label l1 if content of R1 ≠ 0, otherwise to instruction with label l2</td>
</tr>
<tr>
<td>NOP</td>
<td>does nothing</td>
</tr>
</tbody>
</table>

Fig. 8. ASM Semantics for Machine Language

5.2 Basic Code Generation Algorithm

Given an SSA program, machine code is generated in four phases. First, registers are assigned to the results of arithmetic, Boolean, φ, and memory operations. Then φ nodes are eliminated. Afterwards, machine code is generated separately for each basic block. Finally, global code generation unites the generated parts.

Assignment of Registers and Replacement of Operations: First, a fresh result register is assigned to each arithmetic, Boolean, read, and φ operation in the SSA form. Remember that we assume infinitely many registers. Then, these operations in the SSA form are replaced by machine instructions in the obvious way, e.g. add is replaced by ADD. The result of this transformation is a mix between the original SSA form and the machine language. The program is still in its SSA structure but its operations are already replaced by machine instructions.
Elimination of \( \phi \) Nodes: Then, the \( \phi \) operations are eliminated in the standard way: If a block \( b \) contains the \( \phi \) operation \( x = \phi(x_1, \ldots, x_n) \), then in each of the \( n \) predecessor blocks of \( b \), \( x_i \) is copied into the same fresh register \( R: CP(x_i, R) \), \( 1 \leq i \leq n \). Note that the \( x_i \) denote registers because in the first phase, we have assigned registers to the results of operations. In total, there are \( n \) such copy operations, and \( CP(x_i, R) \) is placed in the \( i \)th predecessor block of \( b \). In block \( b \), the \( \phi \) node is replaced by the operation \( CP(R, R') \) for a fresh register \( R' \). For a given basic block \( b \), the result of this phase is denoted by \( E_b \).

Code Generation for Basic Blocks: Afterwards, the arithmetic, Boolean, and memory operations within one basic block are arranged in a linear order which respects the data flow dependencies between them. Since the SSA form specifies only a partial order on the instructions, there are several valid linearizations. Each topological order of the acyclic SSA data flow graph is valid.

Problems arise if the SSA form contains memory write operations whose order is not enforced by the data flow dependencies. For the purpose of this paper, we assume that the write operations are already in a linear order in the SSA form. This assumption is natural because it can easily be met when transforming source programs into SSA form. The source program contains the write operations in a linear order so that the result memory (in the sense of functional stores) of a write operation is the input to the succeeding write operation. The read operations use some results of the write operations. These data flow dependencies determine the partial order between the read and write operations. Each topologic order of them is valid.

Conclusively, basic blocks are transformed into machine code by these steps:

1. First, each basic block \( b \) gets a unique label \( l_b \).
2. Then, the arithmetic, Boolean, memory, and register assignment operations in a basic block \( b \) are sorted topologically into a linear sequence \( S_b \) of machine instructions.
3. Afterwards, the jump or branch operation contained in basic block \( b \) is replaced with a JMP or BRN machine instruction added to the end of \( S_b \). Thereby the labels \( LI, TI \), and \( FI \) are replaced with the labels \( l_{b'} \) assigned to the corresponding basic blocks \( b' \) which are denoted with \( LI, TI, \) and \( FI \).
4. Finally, the instruction \( "l_b : NOP" \) is inserted at the beginning of the sequence \( S_b \) of machine instructions of basic block \( b \).

The complete machine program \( M_p \) is the concatenation of all sequences \( S_b \) for all basic blocks \( b \) in the SSA representation \( p \) such that \( S_{\text{start,block}} \) comes first.

5.3 Correctness of Basic Code Generation Algorithm

To prove the correctness of the code generation algorithm described in subsection 5.2, we distinguish between local and global correctness. Local correctness of code generation means informally that the results computed in a basic block \( b \) are the same, no matter if \( b \) is evaluated according to the SSA semantics or if \( b \) is first
transformed into machine code $S_b$ and then evaluated according to the machine language semantics. Global correctness means that local correctness holds for all basic blocks evaluated during execution. In the remainder of this subsection, we formalize these ideas.

The first step in the code generation algorithm assigns registers to the results of operations in the SSA form. Since we assume infinitely many registers, this assignment is described by an injective function $\text{reg\_assign} : \text{Operations} \rightarrow \text{Registers}$ with the property: \( \text{op}_1 \neq \text{op}_2 \Rightarrow \text{reg\_assign}(\text{op}_1) \neq \text{reg\_assign}(\text{op}_2) \).

Furthermore, the SSA operations are replaced by corresponding machine instructions, described by the injective function $\text{op\_assign} : \{\text{add, AND, read, write, jump, branch}\} \rightarrow \{\text{ADD, AND, READ, WRITE, JMP, BRN}\}$.

**Definition 1 (Local Correctness).** Let $b$ be a basic block of an SSA representation, and let $S_b$ be the machine code generated from it. $b$ and $S_b$ are semantically equivalent if, given that $\text{value}(\text{op}) = \text{reg\_table}(\text{reg\_assign}(\text{op}))$ for all $\text{op} \in \bigcup_{b' \in \text{Preds}(b)} b'$, it follows that for all $\text{op} \in b$, after evaluation of $b$ and $E_b$, $\text{value}(\text{op}) = \text{reg\_table}(\text{reg\_assign}(\text{op}))$ holds.

**Definition 2 (Global Correctness).** Let $p$ be an SSA representation with starting block start\_block and with start\_pred as predecessor block of start\_block. Let $M_p$ be the machine program generated from $p$. $p$ and $M_p$ are semantically equivalent if the following conditions hold:

- $\text{value}(\text{op}) = \text{reg\_table}(\text{reg\_assign}(\text{op}))$ holds for all $\text{op}$ at the beginning of execution of $p$ and $M_p$.
- Execution of $p$ starts with start\_block, execution of $M_p$ starts with $S_{\text{start\_block}}$.
- If the successor of a basic block $b$ is $b'$, then the successor of $S_b$ is $S_{b'}$.

To prove code generation locally and globally correct, we need a formal semantics for the mixed representation still exhibiting the SSA structure but containing already machine instructions. This is achieved with these modifications of the original ASM semantics for SSA:

**Operations:** The SSA operations add, AND, read, write, jump, branch are replaced by $\text{op\_assign}(\text{op})$. E.g. add is replaced by ADD.

**The dynamic function value:** Throughout the ASM specification, the function value is replaced by $\text{reg\_table} \circ \text{reg\_assign}$.

**Evaluation of $\phi$ nodes:** The updates of the values of the $\phi$ nodes are replaced by the updates caused by the evaluation of the replacing copy operations. In general, a basic block might contain two kinds of copy operations, the ones in the beginning which replace the $\phi$ operations directly, and the ones at the end which place the input values for $\phi$ operations in succeeding blocks in the designated registers. When entering a basic block (i.e. $\text{init} = \text{true}$), only the first kind of copy operations is to be evaluated. Hence, we modify the transition rule of figure 5 as follows:
if init and current_block ≠ void then
  for all op ∈ \{ op′ | belongs_to(op′) = current_block ∧ kind(op′) = CP \}
  ∧ kind(Pred_1(op)) = CP \}
  do in parallel value(op) = select(Preds(op), Pred(op)) od;
for all op ∈ \{ op′ | belongs_to(op′) = current_block ∧ (kind(op′) ≠ CP ∨ \)
kind(Pred_1(op)) = CP \} \}
  do in parallel value(op) = undef od;

Lemma 3 ((φ Node Elimination)). Let b be an SSA basic block, $E_b$ the corresponding transformed basic block in the mixed SSA/machine representation, and value(op) = reg_table(reg_assign(op_assign (op))) for all op ∈ \bigcup_{b′ \in Preds(b)} b′. Then value(op) = reg_table(reg_assign(op_assign(op))) holds for all op ∈ b after evaluation of b and $E_b$.

Proof. The proof is by induction. Therefore the operations in b and $E_b$ are partitioned into classes: The first class $C_{1,b}$ of b contains the φ operations of b, the first class $C_{1,E_b}$ of $E_b$ the corresponding CP operations. Given the first 1, \ldots, i classes, the i + 1st is defined as follows: It contains all those operations which can be evaluated if the values of the operations in the classes 1, \ldots, i are known. Because a basic block in SSA form is an acyclic graph, there exists a smallest $n_b$ such that all operations in b are uniquely partitioned into $n_b$ classes. b and $E_b$ are structurally nearly isomorphic: Each operation in b corresponds directly with an operation in $E_b$. Additionally, $E_b$ has some copy operations at the end which place input values for φ operations in succeeding blocks in a special register. These copy operations at the end are placed in the class $C_{n+1,E_b}$.

Base Case: Let op be a φ operation in $C_{1,b}$. Assume that op_assign(op) = CP $R'$, R. By assumption of lemma 3, the predecessors of op have the same values as the predecessors of op_assign(op). The predecessor operations of op_assign(op) in predecessor block $E_{b'}$ are operations CP $R_{b'}$, $R'$ which all copy a value in the same register $R'$. The content of $R'$ is assigned to register R by op_assign(op). This is the same value as one would get when executing the φ operation op in the original SSA form. Since all φ nodes and their substituting register copy operations are executed simultaneously, there is no interference between them. Since all their input values exist, we conclude for all op ∈ $C_{1,b}$, value(op) = reg_table(reg_assign(op_assign(op))) holds after execution of the corresponding transition rule in the original SSA and the modified SSA semantics.

Induction Case: For each class $C_{i,b}$ and its correspondent $C_{i,E_b}$, 2 ≤ i ≤ $n_b$, it follows directly that value(op) = reg_table(reg_assign(op_assign(op))) holds for all op ∈ $C_{i,b}$ after execution of the corresponding transition rules: There is always only one transition rule for each operation, and the value of the operation depends only on its input values which have been determined uniquely in the classes of operations evaluated before. Hence, we conclude for all op ∈ $C_{i,b}$ that value(op) = reg_table(reg_assign(op_assign(op))) holds after execution of the corresponding transition rule in the original SSA and the modified SSA.
semantics by using the induction assumption that for all $op \in \bigcup_{1 \leq j \leq i-1} C_{j,b}$, $\text{value}(op) = \text{reg\_table}(\text{reg\_assign}(\text{op\_assign}(op)))$ which completes the proof.

Note that for the operations in $C_{n+1,E_b}$, the input values are copied to the output values and serve as inputs of the $\phi$ operations in the succeeding block. Their purpose has been discussed already in the base case.

**Lemma 4 ((Local Correctness)).** Let $b$ be an SSA basic block and $S_b$ the corresponding machine code. Then $b$ and $S_b$ are semantically equivalent.

**Proof.** Lemma 4 follows directly from lemma 3: The instructions in $S_b$ are the same as the instructions in $E_b$. The linear order on the instructions in $S_b$ contains the partial order on the instructions in $E_b$, i.e., an instruction is only executed in the schedule of $S_b$ if its operand values have been computed before.

**Theorem 5 ((Global Code Generation)).** Let $p$ be an SSA representation with starting block $\text{start\_block}$ and with $\text{start\_pred}$ as predecessor block of $\text{start\_block}$. Let $M_p$ be the machine program generated from $p$. Then $p$ and $M_p$ are semantically equivalent.

**Proof.** We need to show that the three conditions of definition 2 are fulfilled. The first holds trivially because in the SSA semantics as well as in the machine semantics, all results of operation and values of registers, resp., are initialized with $\text{undef}$, except for the values of operations and registers in $\text{start\_pred}$ and $S_{\text{start\_pred}}$ which are initialized with the input values of the program. The second condition follows in case of the SSA program directly from the ASM semantics for SSA because $\text{current\_block} = \text{start\_block}$ holds in the initial states. In case of the machine semantics, it follows from the fact that $M_p$ starts with $S_{\text{start\_block}}$ and that execution starts with the first instruction.

A simple induction argument shows that the third condition holds for all basic blocks $b$ and $S_b$ executed during the run of $p$ and $M_p$, resp.

**Base Case:** After execution of $\text{start\_block}$ and $S_{\text{start\_block}}$, $\text{value}(op) = \text{reg\_table}(\text{reg\_assign}(\text{op\_assign}(op)))$ holds for all $op \in \text{start\_block}$ (because of lemma 4), and hence for all $op$ in $p$ because execution of $\text{start\_block}$ does not modify operations $op \notin \text{start\_block}$. From the SSA and machine semantics, it follows directly that if the succeeding block in the execution of $p$ is $b'$, then the succeeding block in the execution of $M_p$ is $S_{b'}$. This holds because of the transition rule in figure 7, the algorithm for generating code for connecting basic blocks with JMP/BRN instructions using the labels $LI$, $TI$ and $FI$, and the fact that the results of the branch and BRN as well as of the jump and JMP operations denote corresponding basic blocks in $p$ and $M_p$.

**Induction Case:** Repeat the reasoning in the base case, thereby replacing $\text{start\_block}$ with an arbitrary current block $b$ and $S_{\text{start\_block}}$ with $S_b$.

**5.4 Discussion of Correctness Proof**

The above correctness proof relates the SSA semantics which is partly nondeterministic with the strictly deterministic semantics of the machine language.
Since the SSA semantics puts only a partial order on the operations in a given basic block, several valid linear orders are possible. Our proof shows that each of them is correct if one cares only about block-wise execution and is not interested in the intermediate states reached whenever only a subset of the operations in a basic block has been executed. In this sense, the machine language and the code generation algorithm that we have chosen in this paper, even though being simple, capture the essential problems when proving code generation correct. More advanced code selection algorithms might condense several succeeding operations in basic blocks into single complex machine instructions. This can easily be integrated into our correctness proof because the connection between the overall result of the corresponding subgraph in the SSA block and the result of the complex machine instruction can easily be established. Also code generation for VLIW (very long instruction word) processors can be integrated. In VLIW instructions, several data-independent instructions are executed simultaneously. Since SSA representations explicitly show the data dependencies, candidates for VLIW instructions can easily be identified. (SSA operations are data-independent if there is no directed path between them.) Hence, our correctness proof demonstrates the suitability of the stated ASM semantics for SSA representations and might serve as basis for further correctness proofs of more sophisticated optimizing code generation algorithms. Note that in such optimizing code generation algorithms, it might be necessary to undermine the concept of basic blocks by moving instructions from one basic block to another. Nevertheless, basic blocks are an inherent concept of SSA form and as such, they are part of our formal ASM semantics.

6 Related Work

ASMs have been used for the formal specification of many programming languages, cf. the detailed discussion in section 3. In [ZD03] a specification for the non-deterministic evaluation of expressions based on ASMs has been given which is similar to our specification for the non-deterministic data flow within basic blocks. Moreover, ASMs have been used successfully in proving the correctness of compilations, e.g. the correctness of compiling Prolog to the WAM [BR94], the correctness of translating Occam to transputer code [BD96] and in the Verifix project which deals with the construction of provably correct compilers [ZG97,DV01,DvHG02,GZ99]. While all these compilations are refining transformations, no systematic method is known for proving the correctness of optimizing compilers. In this paper, we have given a necessary prerequisite for such proofs, namely a formal semantics for SSA representations as well as a relatively simple correctness proof for machine code generation extendable to capture also non-refining optimizing transformations.

7 Conclusions

In this paper, we have stated a formal semantics for SSA representations in a simple and elegant way based on ASMs. We have described the state-based
imperative part of SSA computations by transition rules which transfer control flow from the current to the succeeding basic block. Furthermore we have specified the purely data flow driven computation within basic blocks by transition rules which are non-deterministic in the sense that during the evaluation of a basic block more than one set of updates might be applicable. Each specification should demonstrate its usefulness in order to not become an end in itself. We have provided such a demonstration of usefulness by showing that our specification can serve as basis in proofs of correctness for machine code generation.

In future work, we want to prove the correctness of more elaborate code generation algorithms. In particular, we want to extend the machine language to include very long instruction words (VLIW), predicated instructions and speculative execution. This also implies that the code generation algorithm be extended to generate machine code optimized for such instruction sets. Moreover, we want to drop the assumption that there are infinitely many registers by considering optimizing register allocation algorithms as well. Furthermore, we also want to prove the correctness of data flow analyses and corresponding machine independent optimizations of SSA representations. These are optimizations which transform a given SSA form into a semantically equivalent SSA form, e.g. by eliminating dead code or common subexpressions. For all these correctness proofs, the formal SSA semantics and the correctness proof stated in this paper are supposed to serve as basis, as already discussed in subsection 5.4. For many of these optimizations, it is necessary to move instructions between basic blocks. Such transformations will need more sophisticated proof techniques since then, it is harder to identify corresponding states in the original and optimized program.

Acknowledgments

The author would like to thank Wolf Zimmermann and Jan Olaf Blech for many fruitful discussions on the topics of this paper. Also thanks to the anonymous reviewers for their helpful comments.

References


Observations on the Decidability of Transitions

Yuri Gurevich\textsuperscript{1} and Rostislav Yavorskiy\textsuperscript{2,*}

\textsuperscript{1} Microsoft Research
Redmond WA, 98052, USA
gurevich@microsoft.com
\textsuperscript{2} Steklov Mathematical Institute
Gubkina 8, Moscow, 119991, Russia
rey@mi.ras.ru

Abstract. Consider a multiple-agent transition system such that, for some basic types $T_1, \ldots, T_n$, the state of any agent can be represented as an element of the Cartesian product $T_1 \times \cdots \times T_n$. The system evolves by means of global steps. During such a step, new agents may be created and some existing agents may be updated or removed, but the total number of created, updated and removed agents is uniformly bounded.

We show that, under appropriate conditions, there is an algorithm for deciding assume-guarantee properties of one-step computations. The result can be used for automatic invariant verification as well as for finite state approximation of the system in the context of test-case generation from AsmL specifications.

1 Motivating Example

Consider the following simplified model of a file system (a real world file system that the authors were exposed to). Basic information about a file is collected in the File class. For simplicity, we include in the model only very basic file attributes: name and sort. Also, each file keeps a set of the identifiers of its children, and a reference to the parent. Suppose that we want to verify that these references are mutually consistent, that is that every child knows its parent and that every parent knows all the children. We use the syntax of the Abstract State Machines specification language [1,2].

type FileId = Integer
enum FileAttr
  Regular
  Directory
class File
  var fid as FileId // explicit unique identifier
  var sort as FileAttr

* The second author is partially supported by grants from Russian Foundation for Basic Research, Russian Science Support Foundation, and Microsoft Research.
Thus a file system is modeled as a set of file records. The fid field provides an explicit unique identifier of a file. As an object (or class instance), a file is automatically provided with an object ID, but there may be good reasons to have explicit identifiers as well.

The root of the file system has a file ID but no other file fields. A global variable files contains the current set of existing files; initially it is empty. And there is a counter that provides fresh file IDs.

```
const root as FileId = 0
var files as Set of File = {}
var nextFid as FileId = 1
```

We specify common file system operations for creation, deletion and renaming/moving of a resource.

```
procedure Create(parent as File, name as String, sort as FileAttr)
    let fid = nextFid
    let file = new File(fid, sort, name, parent,{},"")
    add file to files
    add fid to parent.children
    nextFid += 1

procedure Delete(file as File)
    let parent = the file' in files where file.parent = file'.fid
    require file.children = {}
    remove file.fid from parent.children
    remove file from files

procedure Move (file as File, newName as String, newParent as File)
    file.parent := newParent.fid
    file.name := newName
    add file.fid to newParent.children
    let oldParent = the file' in files where file.parent = file'.fid
    remove file.fid from oldParent.children
```

The following method formalizes the property to be verified.

```
Invariant() as Boolean
    forall f1 in files, f2 in files holds
        (f1 ne f2) implies (f1.fid in f2.children iff f1.parent = f2.fid)
```

Note that the operations Create and Delete affect two files — all the rest remain unchanged. The Move operation affects only three files, namely the moved file, its old parent and its new parent).

A “manual” proof of the property is simple. Assume that the invariant holds before a transition. One only needs to check that, after the transition is per-
formed, the property holds for the affected files. This splits into several cases. In each cases, checking of the property is easy.

Similar examples arise in modelling of various distributed systems. Usually, in multiple agent transition systems the state of an agent \( a \) is characterized by the values of fields \( a.f_1, \ldots, a.f_N \). Without loss of generality one may assume that all agents have the same set of fields. During one step of the computation some new agents may be created and some previously active agents may be removed or updated. The main restriction we impose is that the set of affected agents — created, removed or updated — is uniformly bounded for all steps.

In the example we checked an invariant. More generally, we may want to check whether a precondition \( \varphi_1 \) at a given state of the system implies a postcondition \( \varphi_2 \) at the next state of the system.

We sought (and found) a decidability result that covers invariant checking and assume-guarantee properties for such systems. The result can be used for automatic invariant verification as well as for finite state approximation of the system in the context of test-case generation from AsmL specifications [3,4].

The rest of the paper is organized as follows. In Section 2 we describe our computation model. In Section 3 we show that under the considered restrictions the assume-guarantee properties of a system are expressible in the first order theory of the underlying structure \( S \). The conclusion is given in Section 4.

## 2 Computational Model

### 2.1 The System State

Let \( I \) denote an infinite index set, that is any countable set with only the equality relation defined on the elements. For simplicity we may assume \( I \) is the set of natural numbers.

Let \( S \) be a structure of signature \( \sigma \). Intentionally, a state of every agent is characterized by a value from \( S \). For the file system example described above the set of elements of \( S \) is

\[
\text{FileVersion} \times \text{FileAttr} \times \text{String} \times \text{FileVersion} \times \text{Set of FileId} \times \text{Boolean}
\]

That is we just take the cartesian product of sets representing types of the class fields. The last element in the product stands for the universe of Boolean values \{true, false\}. We added it because, instead of a variable set of agents, it is convenient to think about a fixed infinite set of agents where the additional Boolean valued field \texttt{Active} indicates whether the record corresponds to a currently existing element or not. The creation of a new agent corresponds to updating the \texttt{Active} field to true for a previously inactive agent. Similarly, when the object is destroyed we just flip the value of the \texttt{Active} field to false.

Let \( f \) be a fresh unary functional symbol. The state of the whole system is characterized by a mapping

\[
f : I \to S.
\]

Namely, agents are identified by elements of \( I \), the state of an agent \( a \) is characterized by the value \( f(a) \).
2.2 The Transition Relation

In what follows \( \text{Diff}_k \) stands for a first order formula asserting that the values of the \( k \) variables are different. For example

\[
\text{Diff}_3(x, y, z) \iff (x \neq y) \land (y \neq z) \land (x \neq z).
\]

Such formulas are expressible in any first-order theory with equality.

In general, the program for a non-deterministic transition \( \tau \) has the following form:

**procedure \( \tau \)**

choose \( i_1, \ldots, i_k \) in \( I \), \( p_1, \ldots, p_m \) in \( S \)

where \( \text{Diff}_k(i_1, \ldots, i_k) \) and \( \delta(f(i_1), \ldots, f(i_k), p_1, \ldots, p_m) \)

\[
f(i_1) := t_1(f(i_1), \ldots, f(i_k), p_1, \ldots, p_m)
\]

\[
\ldots
\]

\[
f(i_k) := t_k(f(i_1), \ldots, f(i_k), p_1, \ldots, p_m)
\]

where, \( \delta \) is a first-order formula over \( \sigma \), and \( t_1, \ldots, t_k \) are terms over \( \sigma \).

Thus one step of the system goes like that: \( k \) different agents are chosen randomly that satisfy the condition formalized by formula \( \delta \). Then, states of the chosen agents are updated accordingly to the program of \( \tau \).

A *computation* is a sequence of states (interpretations of \( f \)) such that each subsequent state is the result of the transition applied to the previous state for a particular choice of the parameters.

2.3 Properties of the Computations

In this paper we are interested in the properties of the following form:

\[
\varphi_1 \to \circ_\tau \varphi_2.
\]

Here \( \circ_\tau \) denotes the well known temporal operator “valid in the next state after transition \( \tau \)” never mind what choices are made by \( \tau \); formulas \( \varphi_1, \varphi_2 \) are of the following form:

\[
\varphi_1 \iff \forall j_1 \ldots j_s \in I \ (\text{Diff}_s(j_1, \ldots, j_s) \to \psi_1(f(j_1), \ldots, f(j_s))),
\]

\[
\varphi_2 \iff \forall j_1 \ldots j_t \in I \ (\text{Diff}_t(j_1, \ldots, j_i) \to \psi_2(f(j_1), \ldots, f(j_i))),
\]

where \( \psi_1, \psi_2 \) are first-order formulas over \( \sigma \) with no free variables.

3 The Main Result

**Theorem 1** For any formulas \( \varphi_1, \varphi_2 \) and transition \( \tau \) as described above the relation \( \varphi_1 \to \circ_\tau \varphi_2 \) is expressible in the first order theory of \( S \).
Proof. First of all, we expand $\varphi_1 \rightarrow \circ_\tau \varphi_2$ according to the definition of $\circ_\tau$. This gives us the following formula:

$$
\forall j_1 \ldots j_s \in I \ (\text{Diff}_s(j_1 \ldots j_s) \rightarrow \psi_1(f(j_1), \ldots, f(j_s))) \rightarrow \\
\forall j_1 \ldots j_t \in I \ (\text{Diff}_t(j_1 \ldots j_t) \rightarrow \psi_2(f'_\tau(j_1), \ldots, f'_\tau(j_t))).
$$

Here $f'_\tau$ stands for the version of $f$ updated according to the transition $\tau$.

Then we expand the definition of $\tau$:

$$
\forall j_1 \ldots j_s \in I \ [\text{Diff}_s(j_1 \ldots j_s) \rightarrow \psi_1(f(j_1), \ldots, f(j_s))] \rightarrow \\
\forall j_1 \ldots j_t \in I \ [\text{Diff}_t(j_1 \ldots j_t) \rightarrow \\
\forall i_1 \ldots i_k \in I \ \forall p_1 \ldots p_m \in S \ [\text{Diff}_k(i_1, \ldots, i_k) \rightarrow \\
\delta(f(i_1), \ldots, f(i_k), p_1, \ldots, p_m) \rightarrow \\
\psi_2(f''(j_1, i, p), \ldots, f''(j_t, i, p))],
$$

where $i, p$ are abbreviations for $i_1, \ldots, i_k$, and $p_1, \ldots, p_m$ correspondingly, and $f''$ is defined in the following way:

$$
f''(j, i, p) = \begin{cases} 
  t_1(f(i_1), \ldots, f(i_k), p_1, \ldots, p_m), & \text{if } j = i, 1 \leq l \leq k; \\
  f(j), & \text{otherwise}.
\end{cases}
$$

Lemma 1 The right hand side of the implication $\varphi_1 \rightarrow \circ_\tau \varphi_2$ is equivalent to a conjunction of formulas of the form

$$
\forall j_1 \ldots j_n \in I [\text{Diff}_n(j_1, \ldots, j_n) \rightarrow \beta(f(j_1), \ldots, f(j_n))]
$$

where $\beta(x_1, \ldots, x_n)$ is a first-order formula over $\sigma$.

Proof. Begin by moving all the universal quantifiers out of the formula. The formula in question acquires the form $\forall j \chi(j)$ where $\chi(j)$ is a boolean combination of equalities $j_p = j_q$ and first-order formulas over $\sigma$ with substituted terms $f(j)$.

To transform this kind of formula to the desired form we apply the following standard procedure.

First, we consider the following tautology: the complete disjunctive normal form where the equalities $j_p = j_q$ play the roles of propositional variables. Every consistent disjunct $\text{Config}(j)$ represents a pattern of equality over the variables $j_p$.

Then, instead of the formula $\chi(j)$, we consider the following implication (which is equivalent to $\chi(j)$ because the antecedent is a tautology):

$$
\bigvee_l \text{Config}_l(j) \rightarrow \chi(j).
$$

This is equivalent to the following conjunction:

$$
\bigwedge_l [\text{Config}_l(j) \rightarrow \chi(j)].
$$
To complete the proof of the lemma we move the universal quantifier over \( j \) inside the conjunction, and then we eliminate all the positive occurrences of equality in \( \text{Config}_1 \). For example:

\[
\forall j_1 j_2 j_3 j_4 (j_1 \neq j_2 \land j_2 = j_3 \land j_1 = j_4 \to \chi(j_1, j_2, j_3, j_4))
\]

is equivalent to

\[
\forall j_1 j_2 (j_1 \neq j_1 \to \chi(j_1, j_2, j_2, j_1)).
\]

Q. E. D.

**Lemma 2** Let \( \alpha(x_1, \ldots, x_k) \) and \( \beta(y_1, \ldots, y_n) \) be two first order formulas in the signature \( \sigma \), where all the free variables of the formulas are explicitly shown. Then the following property

for any function \( f : I \to S \) the following holds:

\[
\forall i_1 \ldots i_k \in I[\text{Diff}_k(i_1 \ldots i_k) \to \alpha(f(i_1), \ldots, f(i_k))] \to \\
\forall j_1 \ldots j_n \in I[\text{Diff}_n(j_1, \ldots, j_n) \to \beta(f(j_1), \ldots, f(j_n))].
\]

is expressible by a first-order formula over \( \sigma \).

**Proof.** The proof follows from the following equivalent transformations.

1. For better readability we replace the text in the first line of the property with the second-order quantifier over \( f \), and then move outside the universal quantifier over \( j \). As the result we get:

\[
\forall f : I \to S, \forall j_1 \ldots j_n \in I[\text{Diff}_n(j_1, \ldots, j_n) \to \\
[\forall i_1 \ldots i_k \in I[\text{Diff}_k(i_1 \ldots i_k) \to \alpha(f(i_1), \ldots, f(i_k))] \to \\
\beta(f(j_1), \ldots, f(j_n))].
\]

2. Observe now, that the property starts with two universal quantifiers: we choose any function \( f \), and then any \( n \) different values of the function arguments. The rest of the formula is a property about values of the function on these arguments. One can easily see that nothing is lost if we just fix values of \( j_1, \ldots, j_n \), e.g. \( j_1 = 1, j_2 = 2, \ldots, j_n = n \).

   Indeed, if the property is true for all values of \( j \) then it is certainly true for these particular values. On the other hand, if it is refuted for some particular choice of \( f \) and \( j \), then one can easily construct \( f' \) by permuting some values of \( f \) in such a way that the property is refuted for \( f' \) and the fixed values of \( j \). So, we can transform to the following:

\[
\forall f : I \to S \\
[\forall i_1 \ldots i_k \in I[\text{Diff}_k(i_1 \ldots i_k) \to \alpha(f(i_1), \ldots, f(i_k))] \to \\
\beta(f(1), \ldots, f(n))].
\]

3. Note now that since \( f \) is arbitrary, values \( f(1), \ldots, f(n) \) are just any values from \( S \). So we get:

\[
\forall a_1 \ldots a_n \in S, \forall f : I \to S \\
[f(1) = a_1 \land \cdots \land f(n) = a_n \to \\
[\forall i_1 \ldots i_k \in I[\text{Diff}_k(i_1 \ldots i_k) \to \alpha(f(i_1), \ldots, f(i_k))] \to \\
\beta(a_1, \ldots, a_n)].
\]
or in disjunctive form:

\[
\forall a_1 \ldots a_n \in S, \forall f : I \to S \\
\neg (f(1) = a_1 \land \cdots \land f(n) = a_n) \lor \exists i_1 \ldots i_k \in I(Diff_k(i_1 \ldots i_k) \land \neg \alpha(f(i_1), \ldots, f(i_k))) \lor \\
\beta(a_1, \ldots, a_n)).
\]

4. Since the last line has no occurrences of \(f\), this is equivalent to:

\[
\forall a_1 \ldots a_n \in S[\beta(a_1, \ldots, a_n)] \lor \\
\forall f : I \to S[(f(1) = a_1 \land \cdots \land f(n) = a_n) \to \\
\exists i_1 \ldots i_k \in I(Diff_k(i_1 \ldots i_k) \land \neg \alpha(f(i_1), \ldots, f(i_k))))]}

5. Without loss of generality one can assume that values of \(i_1 \ldots i_k\) in the last line of the formula are restricted with \(k + n\). Indeed, with this kind of formula we can only distinguish cases when \(i_{\delta} = 1, \ldots, i_{\delta} = n, i_{\delta} > n\), and equalities \(i_{\alpha_1} = i_{\alpha_2}\). In the worst case, after applying the corresponding permutation to \(f\) we get \(i_1 = n + 1, \ldots, i_k = n + k\).

As the result we get the following formula:

\[
\forall a_1 \ldots a_n \forall a_{n+1} \ldots a_{n+k} \in S[\beta(a_1, \ldots, a_n)] \lor \\
\forall f : I \to S[(f(1) = a_1 \land \cdots \land f(n + k) = a_{n+k}) \to \\
\exists i_1 \ldots i_k \in 1, \ldots, n + k(Diff_k(i_1 \ldots i_k) \land \neg \alpha(f(i_1), \ldots, f(i_k))))]
\]

6. Now, the values \(f(i)\) for \(i > n + k\) could be ignored. So instead of a function we can consider sequences of integers:

\[
\forall a_1 \ldots a_{n+k} \in S[\beta(a_1, \ldots, a_n)] \lor \\
\exists i_1 \ldots i_k \in 1, \ldots, n + k(Diff_k(i_1 \ldots i_k) \land \neg \alpha(a_{i_1}, \ldots, a_{i_k}))]
\]

7. To finish the proof note that existential quantifier over the finite set \(\{1, \ldots, n + k\}\) could be replaced with the corresponding finite disjunction.

Q. E. D.

Note that this reduction from the second order language to the first order turns out to be quite simple. It is possible that the result was known, but we don’t know any relevant references.

**Corollary 1** Suppose the first order theory of \(S\) is decidable, then the relation \(\varphi_1 \to \varphi_2\) is decidable too.

**Proof.** Indeed, according to the theorem the relation is expressible by a first order formula over \(S\). So, it is decidable.

## 4 Conclusion

Let \(K\) denote the class of computational systems satisfying the following conditions:
1. The system state is characterized by a finite collections of agents.
2. Each of the agents is characterized by an element of the structure $S$.
3. A transition of the system consists of the following three steps: some new agents arrive, some previously active agents leave the system, and some other agents are updated. The total number of created, updated and removed agents is uniformly bounded.
4. All the updates are expressible by first order formulas over $S$.

**Corollary 2** Suppose the first order theory of $S$ is decidable. Then assume-guarantee properties $\varphi_1 \rightarrow \circ \varphi_2$ of systems from the class $K$ are decidable provided the precondition $\varphi_1$ and the postcondition $\varphi_2$ are expressible by first order formulas over $S$.

One example of $S$ is Presburger Arithmetic of addition [5]; see [6] for more.

**Acknowledgements**

The authors are thankful to anonymous referees for numerous helpful remarks to the text.

**References**

A Security Logic for Abstract State Machines

Stanislas Nanchen and Robert F. Stärk

Computer Science Department, ETH Zürich, CH–8092 Zürich, Switzerland
{nanchen,staerk}@inf.ethz.ch

Abstract. We extend the logic for Abstract State Machines by a read predicate that allows to make precise statements about the accesses of locations of an ASM. The logic can be used to prove security properties of ASMs like that the machine does not read locations containing critical information or that all accesses of the machine are in a well-defined region of the abstract memory. The new read predicate is also useful for proving refinements of parallel ASMs to sequential C-like programs. The logic is complete for hierarchical ASMs and still sound for turbo ASMs. It is integrated in the ASMKeY theorem prover.

1 Introduction

The states of an Abstract State Machine (ASM) are algebraic first-order structures. Algebraic structures are widely used in mathematics but are understood here as a kind of generalized memory. A mathematical term like $f(x + f(y))$ has a similar meaning like an expression $\text{mem}[x + \text{mem}[y]]$ in computer science. The function $f$ is dynamic and can change its values at specific arguments. A pair $\langle f, a \rangle$ is called a location for the function $f$ and the value $f(a)$ is then considered to be the content of the location, like $\text{mem}[a]$ is the content of the memory cell with address $a$. Updating a function means putting a new content into a location of the function. Formally, an update is a pair $\langle l, v \rangle$ where $l$ is a location and $v$ is a value. Dynamic functions are more general than ordinary arrays. They can be of arity greater than one and their arguments can be of any type not just non-negative integers.

When a transition rule of an ASM is evaluated in a given state, it produces a set of updates. If the updates do not conflict, then they are applied to the state and yield the next state in the computation. Hence the change from one state to the next state is fully described by the set of updates. In order to produce the updates, however, a transition rule has also to read some locations of the state. The reading of locations is not observable from outside when we look just at the sequence of states in the computation of an ASM.

In the world of ASMs the states are global states of a system. It has always been assumed that a transition rule can read every location of the state. Therefore the main focus has always been on function updates and the dynamic change of states. The reading of locations has been considered as not so important. The only exception we know is the definition of the micro-steps of an ASM in [2].
When it comes to applications, however, the reading of locations is as important as the updating. It is very important to know that certain processes cannot read certain locations, for example a private key stored somewhere in the memory of a computer or a smart card. Therefore it is very important to have tools and formal systems which allow to prove that programs do not access certain locations or that all accesses of a program are in a certain range of the memory.

In this paper we extend the logic for ASMs introduced in [17]. The new logic allows to express and prove properties like ‘the program $P$ does not read the location $f(x)$’ or ‘whenever program $P$ reads location $f(x)$, then $0 \leq x < 10$’. Thus the logic has a read/access predicate in addition to the already available update predicate. The meaning of the new predicate $\text{acc}(P,f,x)$ is that the program $P$ accesses (reads) the function $f$ at the argument $x$, whereas the old predicate $\text{upd}(P,f,x,y)$ still means that the program $P$ updates the function $f$ at the argument $x$ to the new value $y$.

The logic is designed in a modular way such that a local analysis of a program is sufficient to assess possible vulnerabilities as opposed to a global analysis of all uses of a program. Possible applications of the logic include: applet isolation on smart cards, memory safeness of high-performance code like network filters, elimination of array bound checks, sequentialization of parallel programs.

The logic is implemented as a sequent calculus in the ASMKeY interactive prover. ASMKeY is a derivative of the interactive prover of the KeY Project [1]. The KeY project aims to create an integrated tool for modeling, specifying and verifying object-oriented programs, in particular JavaCard programs.

1.1 Related Work

In [6] dynamic logic (DL), in particular Java Card DL from the KeY Project [1], is used to express the Secure Information Flow problem in a generic program logic. The problem is standardly defined as follows ([5], [10]): given a program $P$ with variables partitioned into two classes, low-security variables $L$ and high-security variables $H$, it is impossible to deduce anything about the initial values of the high-security variables $H$ from the observations of the initial and final values of the low-security variables $L$. A typical DL formula in [6] for expressing the secure information flow with a program $P$, one low-security variable $l$ and one high-security variable $h$ is the following:

$$\forall l \exists r \forall h \langle P \rangle r = l$$

(1)

It asserts that for every initial value of $l$, there exists a final value $r$ such that after termination of $P$ the variable $l$ has the value $r$ independently of the initial value of $h$. Hence, no information flows from $h$ to $l$. As we can see, the main concern of the Secure Information Flow problem is rather leak of information than memory access violation. Therefore, the authors of [6] have no read predicate or equivalent. In Sect. 3.3 we show how the problem of secure information flow can be formalized in the logic for ASMs.
Proof-carrying code [13, 14] is based on the following principles: in order to assure code safety, a code-consumer publishes a security policy that is required to be respected by code-producers. The security policy is given by a Verification Condition Generator (VCGen) for programs with an abstract “machine” of the environment in which the programs run (e.g. a micro-processor). After coding the program, a code-producer uses the VCGen to produce a first-order formula asserting that the program respects the security policy. The code-producer then has to prove that formula and delivers a binary program together with the proof of its correctness. The code-consumer can check the proof and gain trust that the code is secure.

The proof-carrying code approach is a general method applicable from assembler programs to high-level programming languages. However it has been used and implemented mainly for low-level applications, for instance, the safety of kernel extensions [13]. In such setting, the safety policy is defined by a safe interpreter for an assembly language in which the kernel extensions are written. Safe means that the interpreter stops if the programs attempt to issue an instruction that would violate safety.

A safe interpreter is typically implemented by having pre- and post-conditions on execution for each instruction. Important examples are the preconditions for reading and writing to memory which are expressed with the help of two predicates saferd and safewr. These predicates assert that a read to or a write from memory are safe. The VCGen for this security policy is then derived from the safe interpreter.

In [15], Reynolds surveys Separation Logic, an extension of Hoare Logic for expressing and proving properties about low-level programs with shared mutable data structures. The programs for the logic are those of classical Hoare logic with new instructions to allocate, access, write and deallocate a heap (addressable storage). Separation Logic adds to classical Hoare Logic predicates on the heap and, more importantly, a new separation conjunction \( \varphi \circ \psi \) (along with a separation implication), which asserts that \( \varphi \) and \( \psi \) hold for disjoint portions of the heap. This simplifies proofs of sharing properties as showed by several examples and case studies.

In [4], a region-based typing system is combined with an ownership-based typing system for ensuring statically the safety of real-time Java programs. Regions are part of the memory which can be allocated by a thread; it is never reclaimed by the garbage collector, but instead is explicitly destroyed by the thread. Region types allow to verify that programs never follow dangling references. On the other side, ownership types define an ownership hierarchy which allows encapsulation of objects in others. They ensure the absence of illegal access to owned objects and allow effective, modular reasoning about objects. As such, the typing system is not an all-purpose framework for reasoning about general access properties; however, it is linearly decidable, allowing easy and effective integration in real-time Java system with few overhead for the programmer.

Finally the authors of [11] give a solution to the modularity problem of frame properties for object-oriented interface specification languages. Frame properties,
descriptions of which locations a method may modify, are specified with modify clauses. Modify clauses comprise the possible updates but not the accesses to memory.

2 Access Sets of Transition Rules

An ASM consists of a vocabulary \( \Sigma \), an initial state \( A \) for \( \Sigma \), a rule declaration for each rule name and a distinguished rule name of arity zero called the main rule of the machine. A rule declaration for a rule name \( r \) is an expression \( r(x) \triangleq P \), where \( P \) is a transition rule in which there are no free occurrences of variables except of \( x \). Rule declarations can be recursive, i.e., the rule name \( r \) may appear in the body \( P \). We denote the terms of \( \Sigma \) by \( s, t \) and the formulas of \( \Sigma \) by \( \varphi, \psi \). The function names in \( \Sigma \) are declared either as static or dynamic. A term is static, if it does not contain dynamic functions. The transition rules \( P, Q \) are syntactic expressions generated as follows (the function arguments can be read as vectors):

1. Skip Rule: \( \text{skip} \)
   Meaning: Do nothing.
2. Update Rule: \( f(s) := t \)
   Syntax: condition: \( f \) is a dynamic function name of \( \Sigma \)
   Meaning: In the next state, the value of \( f \) at \( s \) is updated to \( t \).
3. Block Rule: \( P \parallel Q \)
   Meaning: \( P \) and \( Q \) are executed in parallel.
4. If Rule: \( \text{if } \varphi \text{ then } P \text{ else } Q \)
   Meaning: If \( \varphi \) is true, then execute \( P \), otherwise execute \( Q \).
5. Let Rule: \( \text{let } x = t \text{ in } P \)
   Meaning: Assign the value of \( t \) to \( x \) and execute \( P \).
6. Forall Rule: \( \text{forall } x \text{ with } \varphi \text{ do } P \)
   Meaning: Execute \( P \) in parallel for each \( x \) satisfying \( \varphi \).
7. Sequence Rule: \( P \text{ seq } Q \)
   Meaning: \( P \) and \( Q \) are executed sequentially, first \( P \) and then \( Q \).
8. Try Rule: \( \text{try } P \text{ else } Q \)
   Meaning: If \( P \) is consistent, execute \( P \), else execute \( Q \).
9. Call Rule: \( r(t) \)
   Meaning: Call transition rule \( r \) with parameters \( t \).

The guards \( \varphi \) in the if-rule and the forall-rule are boolean combinations of equations between terms (quantifier-free formulas). We allow the strict (logical) connectives \( \land \) and \( \lor \) as well as the conditional (short circuit) connectives \( \& \& \) and \( || \) (in AsmL 2, ‘and then’ and ‘or else’, see [7]).

The semantics of transition rules is defined in terms of update sets \( U, V \) and access sets \( A, B \). An access set is a set of accesses. The intended meaning of an access \( (f, a) \) for a state \( A \) is that the machine reads the interpretation of the dynamic function \( f \) at point \( a \). Contrary to the update sets, as we are concerned with read properties only, there is no need of a notion of consistency for access sets.
Definition 1 (Access). An access for \( \mathfrak{A} \) is a location for a dynamic function, i.e. a pair \( \langle f, a \rangle \) where \( f \) is a dynamic function name and \( a \) is an element of the base set of \( \mathfrak{A} \).

Why the restriction of accesses to dynamic functions? There are two main reasons. First, static functions like ‘+’ correspond to the hard-wired functions of a processor. Hence, a program is allowed to evaluate them at every possible argument. If the static function is partial, then the result may be \texttt{undef}. In applications where we want to prove that a program does not generate overflows for arithmetical operations, we have to declare these operations explicitly as dynamic functions. The only consequence will then be that certain principles and rules of the logic are more restricted than they are with static arithmetical functions. Hence, the user has to work more in the proofs.

The second reason is a technical one. If we would consider accesses to static functions, then the substitution principle would be violated. Consider the following valid formula:

\[
\forall x \forall y \ (\text{acc}(f(x) < 0, f, y) \rightarrow y = x)
\]

If we instantiate the variable \( x \) by the term \( f(0) \) then we obtain the following formula that is not true in general, since for the evaluation of \( f(f(0)) \) the function \( f \) has to be read at argument 0 as well as at \( f(0) \):

\[
\forall y \ (\text{acc}(f(f(0)) < 0, f, y) \rightarrow y = f(0))
\]

We have to remember here that the quantifier axioms and the substitution principle of the logic for ASMs in [17] have to be restricted to static terms anyway. For example, the axiom \( \forall x \varphi \rightarrow \varphi[^{x}_{\bot}] \) has to be restricted to static terms \( t \). Consider the following instance of a valid formula \( \forall x \varphi \):

\[
\forall x \ (x = 0 \rightarrow [f(0) := 1]x = 0)
\]

It is not allowed to derive the formula \( \varphi[^{f(0)}_{x}] \) using the non-static term \( f(0) \), since this would yield a non-valid formula:

\[
f(0) = 0 \rightarrow [f(0) := 1]f(0) = 0
\]

Hence, we see that axioms and rules of the logic are already restricted (compared to classical first-order logic). Since we do not want to restrict them further, we do not consider accesses to locations of static functions in Def. 1.

In each step of its computation, an ASM accesses locations of \( \mathfrak{A} \) for producing update sets. To compute these update sets, it has to evaluate terms and formulas that occur in the transition rules. For a term \( t \), we denote by \( \text{AccSet}(t, \mathfrak{A}, \zeta) \) the access set that is needed to compute the value \( [t]^{\mathfrak{A}}_{\zeta} \) in state \( \mathfrak{A} \) with respect to the environment \( \zeta \) that assigns elements of \( \mathfrak{A} \) to the variables of \( t \). The exact definition can be found in Table 1. Note, that if \( t \) is a static term, then its access set is always empty.
Table 1. Access sets of terms.

\[
\text{AccSet}(x, \mathfrak{A}, \zeta) = \emptyset
\]
\[
\text{AccSet}(f(t), \mathfrak{A}, \zeta) = \begin{cases} 
\{\langle f, [t]^{\mathfrak{A}}_\zeta \rangle \} \cup \text{AccSet}(t, \mathfrak{A}, \zeta), & \text{if } f \text{ is dynamic;} \\
\text{AccSet}(t, \mathfrak{A}, \zeta), & \text{otherwise.}
\end{cases}
\]

Table 2. Access sets of rule guards.

\[
\text{AccSet}(s = t, \mathfrak{A}, \zeta) = \text{AccSet}(s, \mathfrak{A}, \zeta) \cup \text{AccSet}(t, \mathfrak{A}, \zeta)
\]
\[
\text{AccSet}(\neg \varphi, \mathfrak{A}, \zeta) = \text{AccSet}(\varphi, \mathfrak{A}, \zeta)
\]
\[
\text{AccSet}(\varphi \star \psi, \mathfrak{A}, \zeta) = \text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta) \quad \text{for } \star \in \{\land, \lor, \rightarrow, \leftrightarrow\}
\]
\[
\text{AccSet}(\varphi \&\& \psi, \mathfrak{A}, \zeta) = \begin{cases} 
\text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta), & \text{if } [\varphi]^{\mathfrak{A}}_\zeta = \text{true;} \\
\text{AccSet}(\varphi, \mathfrak{A}, \zeta), & \text{otherwise.}
\end{cases}
\]
\[
\text{AccSet}(\varphi \mid\mid \psi, \mathfrak{A}, \zeta) = \begin{cases} 
\text{AccSet}(\varphi, \mathfrak{A}, \zeta), & \text{if } [\varphi]^{\mathfrak{A}}_\zeta = \text{true;} \\
\text{AccSet}(\varphi, \mathfrak{A}, \zeta) \cup \text{AccSet}(\psi, \mathfrak{A}, \zeta), & \text{otherwise.}
\end{cases}
\]

Table 2 contains the definition of the access sets that are needed to compute the truth-values of rule guards \( \varphi \). The connectives \&\& and \mid\mid are conditional (short-circuit) operators that do not evaluate the second argument if the result can already be determined by the value of the first argument. Example:

\[
\begin{align*}
\text{AccSet}(x < 2 \&\& f(x) < 2, \mathfrak{A}, \{x \mapsto 1\}) & = \{\langle f, 1 \rangle\} \\
\text{AccSet}(x < 2 \&\& f(x) < 2, \mathfrak{A}, \{x \mapsto 2\}) & = \emptyset
\end{align*}
\]

In the second case the operand \( f(2) < 2 \) is not evaluated, hence the access set is empty.

The semantics of transition rules is inductively defined in Table 3 using a predicate \( \text{yields}(P, \mathfrak{A}, \zeta, U, A) \) with the meaning that the rule \( P \) yields in state \( \mathfrak{A} \) with respect to the environment \( \zeta \) the update set \( U \) and the access set \( A \). Note that, since cyclic rule declarations are allowed, there may be no such sets at all. The update and access sets are unique, since we consider deterministic transition rules only. Non-determinism has to come from outside via monitored functions that are updated by the environment. What is new compared to the definition of the ‘yields’ predicate in [3] is the addition of the access sets \( A, B \). The ‘yields’ predicate has five arguments here whereas it has only four arguments in [3].

By \( \mathfrak{A} + U \) we denote in Table 3 the state obtained from \( \mathfrak{A} \) by firing the updates in \( U \); \( U \oplus V \) denotes the sequential composition of update sets where the updates of \( V \) override possible updates in \( U \). Note that the access sets of the guard \( \varphi \) in a forall-rule have to be computed for every element \( a \in \mathfrak{A} \) and not just for the range \( I \). The range of a formula in the forall-rule is defined as follows:

\[
\text{range}(x, \varphi, \mathfrak{A}, \zeta) = \{v \in [\mathfrak{A}]: [\varphi]^{\mathfrak{A}}_{\zeta[x\mapsto v]} = \text{true}\}
\]
Table 3. The semantics of transition rules.

\[
\begin{align*}
yields(&\text{skip}, \mathcal{A}, \zeta, \emptyset, \emptyset) \\
yields(f(s) := t, \mathcal{A}, \zeta, \{[f, [s]^{\mathcal{A}}], [t]^{\mathcal{A}}\}, A) & \quad \text{where } A = \text{AccSet}(s, \mathcal{A}, \zeta) \cup \text{AccSet}(t, \mathcal{A}, \zeta) \\
yields(P, \mathcal{A}, \zeta, U, A) & \quad \text{yields}(Q, \mathcal{A}, \zeta, V, B) \\
yields(P \par Q, \mathcal{A}, \zeta, U \cup V, A \cup B) & \quad \text{if } [\varphi]^{\mathcal{A}} = \text{true} \\
yields(\text{if } \varphi \text{ then } P \text{ else } Q, \mathcal{A}, \zeta, U, A \cup \text{AccSet}(\varphi, \mathcal{A}, \zeta)) & \quad \text{if } [\varphi]^{\mathcal{A}} = \text{false} \\
yields(\text{let } x = t \text{ in } P, \mathcal{A}, \zeta, U, A \cup \text{AccSet}(t, \mathcal{A}, \zeta)) & \quad \text{where } v = [t]^{\mathcal{A}} \\
yields(\forall x \text{ with } \varphi \text{ do } P, \mathcal{A}, \zeta, \bigcup_{v \in I} U_v, \bigcup_{v \in I} A_v \cup \Phi) & \quad \text{where } I = \text{range}(x, \varphi, \mathcal{A}, \zeta) \text{ and } \Phi = \bigcup_{v \in |x|} \text{AccSet}(\varphi, \mathcal{A}, \zeta[x \mapsto v]) \\
yields(P, \mathcal{A}, \zeta, U, A) & \quad \text{yields}(Q, \mathcal{A} + U, \zeta, V, B) \\
yields(P \text{ seq } Q, \mathcal{A}, \zeta, U \oplus V, A \cup B) & \quad \text{if } U \text{ is consistent} \\
yields(P, \mathcal{A}, \zeta, U, A) & \quad \text{if } U \text{ is inconsistent} \\
yields(\text{try } P \text{ else } Q, \mathcal{A}, \zeta, U, A) & \quad \text{if } U \text{ is consistent} \\
yields(P, \mathcal{A}, \zeta, U, A) & \quad \text{yields}(Q, \mathcal{A}, \zeta, V, B) \\
yields(\text{try } P \text{ else } Q, \mathcal{A}, \zeta, V, A \cup B) & \quad \text{if } U \text{ is inconsistent} \\
yields(P, \mathcal{A}, \zeta, U, A) & \quad \text{where } r(x) \triangleq P \text{ is a rule declaration of } M \\
yields(r(t), \mathcal{A}, \zeta, U, A) & \quad \text{yields}(P, \mathcal{A}, \zeta, U) \\
yields(P, \mathcal{A}, \zeta, U) &
\end{align*}
\]

Example: the rule

\[
\forall x \text{ with } 0 \leq x \&\& x < 10 \&\& 0 < f(x) \text{ do } f(x) := 0
\]

yields the access set \{\langle f, 0 \rangle, \langle f, 1 \rangle, \ldots, \langle f, 9 \rangle\} although the range \(I\) may be a proper subset of \(\{0, 1, \ldots, 9\}\).

3 Extending the Logic for ASMs by an Access Predicate

We extend the basic logic for ASMs introduced in [17] by a new access predicate. The logic is an extension of first-order predicate logic by a modal operator
Table 4. The semantics of the modal operator and the basic predicates.

\[
\begin{align*}
\[ [P] \varphi \]^{A}_{\zeta} &= \begin{cases} 
true, & \text{if } \mathcal{I}_{\zeta}^{A+U} = true \text{ for each consistent update set } U \text{ such that } yields(P, A, \zeta, U) \text{ is derivable in Table 3;} \\
false, & \text{otherwise.}
\end{cases} \\
\[ \text{def}(P) \]^{A}_{\zeta} &= \begin{cases} 
true, & \text{if there exists an update set } U \text{ such that } yields(P, A, \zeta, U) \text{ is derivable in Table 3;} \\
false, & \text{otherwise.}
\end{cases} \\
\[ \text{Con}(P) \]^{A}_{\zeta} &= \begin{cases} 
true, & \text{if there exists a consistent update set } U \text{ such that } yields(P, A, \zeta, U) \text{ is derivable in Table 3;} \\
false, & \text{otherwise.}
\end{cases} \\
\[ \text{upd}(P, f, s, t) \]^{A}_{\zeta} &= \begin{cases} 
true, & \text{if there exists an update set } U \text{ such that } yields(P, A, \zeta, U) \text{ and } \langle (f, [s]^{A}_{\zeta}), [t]^{A}_{\zeta} \rangle \in U; \\
false, & \text{otherwise.}
\end{cases} \\
\[ \text{inv}(P, f, s) \]^{A}_{\zeta} &= \begin{cases} 
true, & \text{if there exists an update set } U \text{ such that } yields(P, A, \zeta, U) \text{ and } \langle (f, [s]^{A}_{\zeta}), v \rangle \notin U \text{ for all } v \in |A|; \\
false, & \text{otherwise.}
\end{cases}
\end{align*}
\]

and several atomic predicates. The formulas of the logic are generated by the following grammar:

\[
\varphi, \psi ::= s = t \mid \neg \varphi \mid \varphi \land \psi \mid \varphi \lor \psi \mid \varphi \rightarrow \psi \mid \forall x \varphi \mid \exists x \varphi \mid [P] \varphi \mid \text{def}(P) \mid \text{Con}(P) \mid \text{inv}(P, f, s) \mid \text{acc}(t, f, s) \mid \text{acc}(\varphi, f, s) \mid \text{acc}(P, f, s)
\]

The formula \([P] \varphi\) means that, if \(P\) is defined and consistent, then \(\varphi\) is true in the next state after executing \(P\); \(\text{def}(P)\) means that \(P\) is defined and updates the dynamic function \(f\) at the argument \(s\) to the new value \(t\); \(\text{Con}(P)\) asserts that \(P\) is defined and yields a consistent update set; \(\text{Con}(P)\) asserts that \(P\) is defined and yields a consistent update set (strong consistency); \(\text{inv}(P, f, s)\) means that \(P\) is defined and does not update \(f\) at argument \(s\). The new predicates \(\text{acc}(t, f, s)\) and \(\text{acc}(\varphi, f, s)\) assert that the term \(t\) resp. the formula \(\varphi\) accesses the function \(f\) at the argument \(s\). The predicate \(\text{acc}(P, f, s)\) asserts that \(P\) is defined and accesses \(f\) at the argument \(s\).

If \(\varphi\) is a rule guard, then by \(\hat{\varphi}\) we denote the formula that is obtained from \(\varphi\) by replacing the conditional \(\&\&\) by the logical conjunction \(\land\) and the conditional \(||\) by the logical disjunction \(\lor\). From the logical point of view the formula \(\hat{\varphi}\) is equivalent to \(\varphi\). From the operational point of view, however, they are different. The formula \(\hat{\varphi}\) accesses more locations than \(\varphi\).

The exact semantics of the modal operator and the basic predicates already present in [17] is given in Table 4. The semantics of the access predicate for terms, formulas and transition rules is given in Table 5. Note, that some of the predicates can be defined in terms of others. For example, the strong consistency predicate can be defined as follows (with fresh variables \(x, y, z\)):

\[
\text{Con}(P) \leftrightarrow \text{def}(P) \land \bigwedge_{f \text{ dyn.}} \forall x, y, z (\text{upd}(P, f, x, y) \land \text{upd}(P, f, x, z) \rightarrow y = z)
\]
The invariance predicate can be defined in terms of other predicates, too:

\[ \text{inv}(P, f, s) \leftrightarrow \text{def}(P) \land \forall x \neg \text{upd}(P, f, s, x) \]

### 3.1 Axioms and Rules of for the Extended Logic

The extended logic \( L^{\text{acc}}(M) \) for an ASM \( M \) comprises in addition to the axioms and rules of the basic system of [17] (which can also be found in [3, p. 319]) the following axioms:

**X. Axioms for acc:**

18. Axioms AT1–AT3 in Table 6
19. Axioms AF1–AF5 in Table 7
20. Axioms AR1–AR9 in Table 8
21. \( \text{acc}(P, f, x) \rightarrow \text{def}(P) \)
22. \( \text{Con}(P) \land \text{def}(Q) \land \text{not_read}(P, Q) \rightarrow \text{same_after}(P, Q) \land [P]\text{def}(Q) \)

The predicate ‘not_read’ means that \( Q \) does not read locations updated by \( P \):

\[ \text{not_read}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \forall x (\text{acc}(Q, f, x) \rightarrow \text{inv}(P, f, x)) \]

The predicate ‘same_after’ means that the updates and the reads of \( Q \) are the same after executing \( P \):

\[ \text{same_after}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \left\{ \forall x, y (\text{upd}(Q, f, x, y) \leftrightarrow [P]\text{upd}(Q, f, x, y)), \forall x (\text{acc}(Q, f, x) \leftrightarrow [P]\text{acc}(Q, f, x)) \right\} \]

The notions of logical consequence and formal derivability are defined as usual. If \( M \) is an ASM and \( \Psi \) a set of sentences, then \( \Psi \models_M \varphi \) means that \( \varphi \) is true in every structure that makes all formulas in \( \Psi \) true; \( \Psi \vdash_M \varphi \) means that there exists a finite subset \( \Theta \subseteq \Psi \) such that the formula \( \bigwedge \Theta \rightarrow \varphi \) is derivable using the axioms and rules of the logic.

In order to show that the logic is still sound one has to verify that the new axioms in Tables 6–8 are valid. For Axiom 22 the following lemma is used.

**Lemma 1.** If \( \text{yields}(P, \mathfrak{A}, \zeta, U, A) \) and \( W \) is a consistent update set that does not contain updates for locations in \( A \), then \( \text{yields}(P, \mathfrak{A} + W, \zeta, U, A) \).

**Proof.** Note first that the values of terms and rule guards depend only on the locations in their access sets. Let \( W \) be a consistent update set that does not update locations in \( \text{AccSet}(t, \mathfrak{A}, \zeta) \). Then we have

\[ [t]^\mathfrak{A}\zeta = [t]^{\mathfrak{A}+W}\zeta \quad \text{and} \quad \text{AccSet}(t, \mathfrak{A}, \zeta) = \text{AccSet}(t, \mathfrak{A} + W, \zeta). \]

A similar property holds for rule guards.

The lemma can then be proved by induction on the definition of the ‘yields’ predicate in Table 3. \( \square \)
Table 5. The semantics of the acc predicate.

\[
\begin{align*}
[\text{acc}(t, f, s)]_{\zeta}^3 & = \begin{cases} 
true, & \text{if } \langle f, [s]_{\zeta}^3 \rangle \in \text{AccSet}(t, \mathfrak{A}, \zeta); \\
false, & \text{otherwise.}
\end{cases} \\
[\text{acc}(\varphi, f, s)]_{\zeta}^3 & = \begin{cases} 
true, & \text{if } \langle f, [s]_{\zeta}^3 \rangle \in \text{AccSet}(\varphi, \mathfrak{A}, \zeta); \\
false, & \text{otherwise.}
\end{cases} \\
[\text{acc}(P, f, s)]_{\zeta}^3 & = \begin{cases} 
true, & \text{if there exist } U \text{ and } A \text{ with } \text{yields}(P, \mathfrak{A}, \zeta, U, A) \\
\text{and } \langle f, [s]_{\zeta}^3 \rangle \in A; \\
false, & \text{otherwise.}
\end{cases}
\end{align*}
\]

Table 6. Axioms of the acc predicate for terms.

AT1. \( \neg \text{acc}(y, f, x) \)
AT2. \( \text{acc}(f(t), f, x) \leftarrow x = t \lor \text{acc}(t, f, x) \)
AT3. \( \text{acc}(g(t), f, x) \leftarrow \text{acc}(t, f, x) \) \quad \text{if } f \neq g

Table 7. Axioms of the acc predicate for rule guards.

AF1. \( \text{acc}(s = t, f, x) \leftarrow \text{acc}(s, f, x) \lor \text{acc}(t, f, x) \)
AF2. \( \text{acc}(\neg \varphi, f, x) \leftarrow \text{acc}(\varphi, f, x) \)
AF3. \( \text{acc}(\varphi \star \psi, f, x) \leftarrow \text{acc}(\varphi, f, x) \lor \text{acc}(\psi, f, x) \) \quad \text{where } \star \in \{\land, \lor, \rightarrow\}
AF4. \( \text{acc}(\varphi \& \& \psi, f, x) \leftarrow \text{acc}(\varphi, f, x) \lor (\check{\varphi} \land \text{acc}(\psi, f, x)) \)
AF5. \( \text{acc}(\varphi \mid \mid \psi, f, x) \leftarrow \text{acc}(\varphi, f, x) \lor (\neg \check{\varphi} \land \text{acc}(\psi, f, x)) \)

Table 8. Axioms of the acc predicate for transition rules.

AR1. \( \neg \text{acc}(	ext{skip}, f, x) \)
AR2. \( \text{acc}(g(s) := t, f, x) \leftarrow \text{acc}(s, f, x) \lor \text{acc}(t, f, x) \)
AR3. \( \text{acc}(P \text{ par } Q, f, x) \leftarrow \text{def}(P \text{ par } Q) \land (\text{acc}(P, f, x) \lor \text{acc}(Q, f, x)) \)
AR4. \( \text{acc}(\text{if } \varphi \text{ then } P \text{ else } Q, f, x) \leftarrow \text{def}(\text{if } \varphi \text{ then } P \text{ else } Q) \land (\text{acc}(\varphi, f, x) \lor (\check{\varphi} \land \text{acc}(P, f, x)) \lor (\neg \check{\varphi} \land \text{acc}(Q, f, x))) \)
AR5. \( \text{acc}(\text{let } y = t \text{ in } P, f, x) \leftarrow \text{def}(\text{let } y = t \text{ in } P) \land (\text{acc}(t, f, x) \lor \exists y (y = t \land \text{acc}(P, f, x))) \) \quad \text{if } y \notin \text{FV}(t)
AR6. \( \text{acc}(\text{forall } y \text{ with } \varphi \text{ do } P, f, x) \leftarrow \text{def}(\text{forall } y \text{ with } \varphi \text{ do } P) \land \exists y (\text{acc}(\varphi, f, x) \lor (\check{\varphi} \land \text{acc}(P, f, x))) \)
AR7. \( \text{acc}(P \text{ seq } Q, f, x) \leftarrow (\text{acc}(P, f, x) \land [P]\text{def}(Q)) \lor (\text{Con}(P) \land [P]\text{acc}(Q, f, x)) \)
AR8. \( \text{acc}(\text{try } P \text{ else } Q, f, x) \leftarrow (\text{acc}(P, f, x) \land (\text{Con}(P) \lor \text{def}(Q))) \lor (\text{def}(P) \land \neg \text{Con}(P) \land \text{acc}(Q, f, x)) \)
AR9. \( \text{acc}(r(t), f, x) \leftarrow \text{acc}(P^+_r, f, x) \) \quad \text{if } r(x) \triangleq P \text{ is a rule declaration of } M
Theorem 1 (Soundness of the logic). If $\Psi \vdash_M \varphi$, then $\Psi \models_M \varphi$.

An ASM is called \textit{hierarchical} if it does not contain cycles in the dependency graph of rule declarations. Hierarchical ASMs could also be called \textit{recursion-free} ASMs. The completeness proof in [17] for hierarchical ASMs can then be extended to the $\text{acc}$ predicate. It can even be shown that the logic is a definitional extension of FOL.

Theorem 2 (Completeness of the logic). Let $M$ be a hierarchical ASM. If $\Psi \models_M \varphi$, then $\Psi \vdash_M \varphi$.

The Axiom 22 is not used in the completeness proof. In case of hierarchical ASMs it can be derived from the other axioms.

3.2 Application: Sequentialization of ASMs

If ASMs are refined to programs of a traditional imperative language, then all parallel compositions have to be sequentialized. The following principle is useful to prove the correctness of such refinement steps. We first define what it means that ‘$Q$ does not overwrite updates of $P$’:

$$\text{not\_over}(P, Q) \triangleq \bigwedge_{f \text{ dyn.}} \forall x, y (\text{upd}(P,f,x,y) \rightarrow \text{inv}(Q,f,x) \lor \text{upd}(Q,f,x,y))$$

The following sequentialization principle can be derived from Axiom 22:

$$\text{Con}(P) \land \text{def}(Q) \land \text{not\_read}(P, Q) \land \text{not\_over}(P, Q) \rightarrow (P \text{ par } Q) \equiv (P \text{ seq } Q)$$

It says, that under the above conditions, the parallel composition of $P$ and $Q$ is equivalent to the sequential composition, i.e. it yields the same updates and reads the same locations. Other approaches to sequentialization of ASMs can be found in [8, 16].

3.3 Memory Access vs. Information Flow

Let $l$ be a location of a dynamic function in a state $\mathfrak{A}$. We say that a transition rule $P$ does not depend on the location $l$ in $\mathfrak{A}$, if $P$ yields the same update set whatever the content of $l$ is. More precisely, if $P$ yields the update set $U$ in state $\mathfrak{A}$ under $\zeta$, then for every element $v \in |\mathfrak{A}|$, $P$ yields the same set $U$ in the state $\mathfrak{A} + \{(l, v)\}$ under $\zeta$.

It is intuitively clear that if a program does not access a location $l$, then it does not depend on the location $l$. The converse is not true. A program may read a location $l$ and simply ignore the content of $l$.

That the updates of a program $P$ do not depend on the location $f(x)$ can be expressed in the logic as follows (where $a, b, y$ are fresh variables):

$$\text{not\_dep\_upd}(P, f, x) \triangleq \forall a, b, y (\text{upd}(P,f,a,b) \leftrightarrow [f(x) := y]\text{upd}(P,f,a,b))$$
The following formula is then a special case of Axiom 22:

\[
\text{def}(P) \land \neg \text{acc}(P, f, x) \rightarrow \text{not\_dep\_upd}(P, f, x)
\]  

(2)

Since the logic is complete for hierarchical ASMs the above principle is derivable from the basic axioms and rules in case of hierarchical ASMs.

We have defined that a program does not depend on a location \( f(x) \), if it produces the same update sets whatever the content of \( f(x) \) is. A weaker notion would be to require that the contents of the locations different from \( f(x) \) after executing the program do not depend on the content of \( f(x) \). In other words, the result of applying the updates does not depend on \( f(x) \). This can be expressed in the logic as follows:

\[
\text{not\_dep\_step}(P, f, x) \triangleq \forall a \,(a \neq x \rightarrow \exists b \forall y \,[f(x) := y][P][f(a) = b])
\]

The two notions are different, since a program can produce trivial updates that do not change the contents of locations. For example, the updates of the program

\[
\text{if } f(0) = 0 \text{ then } f(1) := f(1)
\]

depend on the location \( f(0) \) but the next state is always identical to the present state. The following implication, however, is valid:

\[
\text{not\_dep\_upd}(P, f, x) \rightarrow \text{not\_dep\_step}(P, f, x)
\]  

(3)

The formula can easily be derived already in the basic logic of [17].

The formula \( \text{not\_dep\_step}(P, f, x) \) is the translation of the information flow formula 1 of Sect. 1.1 into the logic for ASMs. The location \( f(x) \) is a high-security variable whereas the locations \( f(a) \) with \( a \neq x \) are low-security variables. It is not possible to translate formula 1 directly, since in DL variables can be updated by programs whereas in the logic for ASMs variables are read-only variables that cannot be updated by transition rules.

That a program does not depend on a location \( l \) should also include that the termination of the program and the consistency of the computed update sets do not depend on \( l \). Otherwise, an adversary could get information on \( l \) based on error messages about conflicting updates. That the termination and consistency of \( P \) do not depend on the location \( f(x) \) can be expressed as follows:

\[
\text{not\_dep\_con}(P, f, x) \triangleq \forall y \,(\text{Con}(P) \leftrightarrow [f(x) := y]\text{Con}(P))
\]

Remember that the predicate \( \text{Con}(P) \) includes the termination of \( P \) (strong consistency).

Joshi and Leino’s semantic definition of secure information flow of [10] can be applied to ASMs as follows:

\[
\text{JoshiLeino}(P, f, x) \triangleq \forall y \,((P \text{ seq } H) \simeq ((H \text{ seq } P) \text{ seq } H))
\]

where \( y \) is not free in \( P \) and \( H \) is the program \( f(x) := y \) that updates \( f(x) \) to an arbitrary value \( y \). The symbol \( \simeq \) means that the transition rules are equivalent. It can be defined as follows (see [17]):

\[
P \simeq Q \triangleq (\text{Con}(P) \leftrightarrow \text{Con}(Q)) \land \forall a \forall b \,([P][f(a) = b] \leftrightarrow [Q][f(a) = b])
\]
The relationship between Joshi and Leino’s definition and the predicates we have introduced so far is as follows:

\[
\text{JoshiLeino}(P, f, x) \leftrightarrow \text{not} \_ \text{dep} \_ \text{con}(P, f, x) \land \text{not} \_ \text{dep} \_ \text{step}(P, f, x)
\]  

(4)

This equivalence is valid for arbitrary transition rules \(P\) and can be derived in the basic logic.

3.4 Access Sets and Critical Terms

In [9], Gurevich derives the sequential ASM thesis from three postulates about sequential algorithms. The third postulate, the uniformly-bounded-exploration postulate, says that an algorithm \(A\) examines only a bounded number of elements in any state. There must exist a finite set \(T\) of ground terms — depending on the algorithm only and not on the initial state — such that the next computation step of the algorithm depends only on that part of the state which can be accessed via terms in \(T\). Whenever two states \(\mathcal{A}\) and \(\mathcal{B}\) coincide over \(T\), then \(\Delta(A, \mathcal{A}) = \Delta(A, \mathcal{B})\), where \(\Delta(A, \mathcal{A})\) is the set of non-trivial updates that have to be applied to state \(\mathcal{A}\) in order to obtain the next state in the computation of the algorithm \(A\). The terms in \(T\) are called critical terms of \(A\).

What can be said on the relationship between critical terms and read/access of locations? In general only sequential ASMs satisfy the uniformly-bounded-exploration postulate. In our setup these are hierarchical ASMs that do not contain the forall-rule. Only for this class of ASMs the existence of a finite set of critical terms is guaranteed.

A first but obviously wrong conjecture could be that the set of locations accessed by a sequential ASM is contained in \(T\). More precisely, that if the location \(\langle f, a \rangle\) is read by the ASM, then there exists a critical term \(f(s)\) where \(a\) is the value of \(s\). This is not true as the following example shows:

\[
\text{if } f(0) = 0 \text{ then } f(1) := f(1) \text{ else } f(1) := f(1)
\]

In this case, the empty set is a possible candidate for the set of critical terms, but the locations \(f(0), 0, 1\) that are accessed by the program are not in this set. The problem is that the program reads the locations \(f(0), 0, 1\) but does not depend on them.

The second conjecture, that the locations a program depends on are contained in the set of critical terms, is also wrong. A possible set of critical terms for the program

\[
\text{if } f(f(0)) = 0 \text{ then } f(0) := 1
\]

is \(T = \{0, 1, f(f(0))\}\). Obviously the program depends on \(f(0)\) but the term \(f(0)\) is not in \(T\) (it is however a subterm of a term in \(T\)).

3.5 Maps in AsmL

In AsmL 2, a map \(f\) is a partial function and the application \(f(x)\) throws an exception if the argument \(x\) is not in the domain of \(f\). In the following example, an empty function \(\mathbf{f}\) is declared. When \(\mathbf{f}\) is applied to the argument \(3\), an \texttt{IndexOutOfBoundsException} is thrown:
\[ \forall x (\text{acc}(P, f, x) \rightarrow f(x) \neq \text{undef}) \]

The formula asserts that \( P \) only accesses defined locations of dynamic functions. Hence no \texttt{IndexOutOfBoundsException} will be thrown when \( P \) is evaluated.

In AsmL 2 it is possible to test whether an element \( x \) belongs to the domain of a map \( f \) using the expression \texttt{‘x in f’}. The question now is whether \texttt{‘x in f’} should be considered as an access of the location \( f(x) \) or not. If \texttt{‘x in f’} is translated as \( f(x) \neq \text{undef} \), then it is definitely an access according to our definitions in Tables 1 and 2. If \texttt{‘x in f’} is not considered to be an access of \( f(x) \), then one has to extend the language of the logic and allow also atomic formulas \texttt{‘t in f’}, where \( \text{AccSet}(t \text{ in } f, \mathfrak{A}, \zeta) = \text{AccSet}(t, \mathfrak{A}, \zeta) \).

### 3.6 Access and Definedness

It is not possible to make statements in our logic about the accesses of non-terminating transition rules. For example, if the rule

\[ r(x) \triangleq \text{if } 0 < f(x) \text{ then } r(x + 1) \]

is called with \( r(0) \), it might not terminate and access an infinite number of locations of \( f \). We think that this is no problem, since the single transitions of ASMs are always terminating. Otherwise, there is something wrong with the modeling of the system. Every proof of an ASM should therefore start with an invariant \( \varphi \) that implies the termination and consistency of the machine: \( \varphi \rightarrow \text{Con}(M) \land [M] \varphi \).

### 4 Implementation in ASMKeY

ASMKeY [12] is a theorem prover for ASMs derived from the KeY Project [1] theorem prover. The ASMKeY prover is based on the logic for ASMs of [17]. The underlying sequent calculus has been extended by the new \texttt{acc} predicates. The new available properties are now applied in case studies, in particular the memory access properties (violation and non-violation) and the sequentialization properties.

To illustrate these two kinds of properties, we will use the \texttt{MERGESORT} algorithm in Fig. 1. It uses the short-circuit operators \&\& and |\|. As opposed to \texttt{MERGESORT}, we will refer by \texttt{MERGESORTV} (for violation) to the variant with the usual \& and |.
\textbf{MergeSort}$(l, r) = $
\begin{align*}
\text{if } l < r \text{ then} & \\
\quad \text{let } m = \lfloor (l + r) / 2 \rfloor \text{ in} & \\
& (\text{MergeSort}(l, m) \parallel \text{MergeSort}(m + 1, r)) \text{ seq} \\
& \text{Merge}(l, m, r)
\end{align*}

\text{Merge}(l, m, r) =
\begin{align*}
\text{(forall } i \text{ with } l \leq i \leq r \text{ do } g(i) := f(i)) \text{ seq} & \\
& \text{MergeCopy}(l, m, m + 1, r, l)
\end{align*}

\text{MergeCopy}(i, m, j, r, k) =
\begin{align*}
\text{if } k \leq r \text{ then} & \\
\quad \text{if } (i \leq m \land j \leq r \land g(i) \leq g(j)) \land (r < j) \text{ then} & \\
& f(k) := g(i) \parallel \text{MergeCopy}(i + 1, m, j, r, k + 1) \\
\quad \text{else} & \\
& f(k) := g(j) \parallel \text{MergeCopy}(i, m, j + 1, r, k + 1)
\end{align*}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig1.png}
\caption{The MergeSort algorithm.}
\end{figure}

\subsection{Memory Access}

Although we had formally proved using ASMKeY that \text{MergeSortV} is correct, at run-time, in AsmL, it generated an \text{IndexOutOfBoundsException}: the dynamic function \(g\) was being accessed at \(r + 1\). Actually, the violating access was made during the computation of the second condition of the \text{MergeCopyV} rule in the test \(g(i) \leq g(j)\).

It is easy to see that, if there exists an index \(i\) in the first half of the array, where \(f(i)\) is greater than \(f(j)\) for every \(j\) in the second half of the array, then a memory violation will occur. In fact, one can see that, in such a case, the merging of the two sorted subarrays will consume at first the second subarray and will cause the formula \(g(i) \leq g(j)\) to be evaluated with \(j = r + 1\). One can even formally prove the following formula:

\[
\forall l, r, m\ (l < r \land m = (l + r)/2 \\
\land \exists i\ (l \leq i \leq m \land \forall j\ (m + 1 \leq j \leq r \rightarrow f(j) < f(i))) \\
\rightarrow \text{acc(MergeSort}(l, r), g, r + 1))
\]

If we use \text{MergeSort} instead of \text{MergeSortV}, then we can prove that there are no accesses of the dynamic function \(g\) outside the interval \([l, r]\):

\[
\forall l, r, x\ (\text{acc(MergeSort}(l, r), g, x) \rightarrow l \leq x \land x \leq r)
\]

For the dynamic function \(f\), it is easy to see that, in \text{MergeSort} as well as in \text{MergeSortV}, we have never access violation, since \(f\) is only accessed during the copy of \(f\) into \(g\).

\subsection{Sequentialization}

Using the principle of section 3.2, we can prove in ASMKeY that \text{MergeSort} and its sequentialization \text{SEQMergeSort} are equivalent:
\[ \forall l, r (\text{MergeSort}(l, r) \simeq \text{SeqMergeSort}(l, r)) \]

The \text{SeqMergeSort} is obtained from \text{MergeSort} as a refinement by replacing every \text{par} rule construct by \text{seq}. It is basically a C version of \text{MergeSort}.

5 Conclusion

We have presented an extension of the basic logic for ASMs of [17] that allows to prove also access and read properties of ASMs. One open problem that remains is to extend the logic to ASMs with the choose-rule and still keep it complete for a large class of ASMs.

References


Slicing Abstract State Machines

Antje Nowack

Mathematische Grundlagen der Informatik
RWTH Aachen, D-52056 Aachen, Germany
nowack@informatik.rwth-aachen.de

Abstract. In this work, we introduce slicing for Abstract State Machines (ASMs). The idea of this concept is analogous to the one of program slicing which is an established technique for extracting statements from a program that are relevant for its behaviour at a given point of interest. These statements form again a syntactically correct program called a slice. Previous work has focused on programming languages that differ substantially from ASMs. Although the concept of program slicing does not directly extend to ASMs, it is possible to find an analogous concept for ASMs. We present such an approach.

In spite of the fact that a minimal slice is not computable in the general case, we identify an expressive class of ASMs for which a minimal (static) slice is computable (and prove the computability). This basic result can be extended in several ways. We present some extensions to larger classes of ASMs and other variants for the notion of slicing.

1 Introduction

In order to debug or to modify a program, we are most often not interested in its complete behaviour but only in its behaviour at a given point of interest. This means that we do not observe each complete program state but only a part of it. Consider for instance the case of debugging. When an error is observed, the programmer tries to extract that part of the program which is responsible for the erroneous behaviour. This set of statements might be much smaller than the original one and therefore, it might be much easier to catch and correct the error(s) if he knows this set.

The above observation motivates the concept of program slicing. By applying this established technique, one extracts statements from a program that are relevant for the behaviour of the program at some specified point of interest. The result is a program slice which forms a syntactically correct program again.

Program slicing has first been considered by M. Weiser [6]. However, his considerations are only based on block-structured, possibly recursive programs written in a Pascal-like language. Weiser defined the notion of slice as follows. Let $P$ be a program, $p$ be the label (e.g. an index) of a statement in $P$ and $V$ be a subset of the program variables of $P$. We are interested in the (sequence of) values assigned to $V$ just before the statement with label $p$ is executed. A (static) slice of $P$ relative to the slicing criterion $\langle p, V \rangle$ is obtained from $P$ by removing statements in such a way that the values of the variables in $V$
just before executing the statement with label $p$ are the same for the complete program $P$ and its slice.

As every program is a slice of itself, it is clear that a slice does always exist. Furthermore, there are usually many slices of a program. One can say that the smaller a slice the better. Therefore, it would be ideal to have a minimal slice of a program. Unfortunately, minimal slices are not automatically computable in general.

There has been extensive research on program slicing, on automatic computation of slices, and on methods for approximating minimal slices. See [5] for an overview.

Abstract State Machines (ASMs) are a formal model of computation as well as an executable specification language used in practice. This suggests to combine practically established concepts with theoretically founded approaches for validating ASM specifications.

One technique widely spread in practice is to validate programs via testing. The executability implies that testing can also be applied to ASMs. Testing becomes more efficient and more accurate if it is appropriately combined with theoretically sound concepts. In this sense, the concept of slicing is a good supplement for testing assumed that we have a formal basis for slicing. Another approach that could be combined with slicing is to generate test cases from an ASM specification in order to check an implementation against its specification on these test cases. This has been realized in the AsmL Test Generator tool developed by Microsoft Research. Another approach has been presented in [2]. Moreover, there are many other fields where a combination with slicing has an advantageous effect. For example, it would be reasonable to combine it with verification (regardless whether manual, semi-automatic, automatic) or to use it for modifying existing ASMs.

The contribution of this work can be summarized as follows.

**Formal Introduction of Slicing for ASMs.** The design of the programming languages that are originally considered for slicing differs substantially from the one of ASMs such that the original concept of program slicing does not extend directly to ASMs. In spite of these differences, it is possible to find a concept for ASMs that is analogous to the one of program slicing. We present such an approach.

**A computability Result.** It is easy to see that a minimal (static) slice cannot be computed automatically for unrestricted ASMs. However, we present a class of ASMs for which a minimal slice is computable (and prove the computability). In addition, it turns out that this class has a high expressive power. It captures a number of expressive formalisms well-known from literature.

**Extensions.** Once we have proven the basic computability result it is easy to obtain several extensions of it. We present some of them. First, we extend the class and explain how the basic result extends to these superclasses. Afterwards we explain how one can vary the notion of slice and how the basic result can be adapted to these variations.
2 Definition of Slicing

As this paper is written for the ASM community, we assume that the reader is familiar with ASMs. For a detailed introduction to ASMs see e.g. [4].

If no restriction is given then the guard of an if-clause, forall-rule or choice-rule may be any first-order formula (possibly containing quantifiers).

In this section, we formally introduce the notion of slice for ASMs. At the beginning, we define the notion of slicing criterion characterizing those parts of a state in which we are interested. Afterwards we formalize what it means to remove statements of an ASM in order to obtain a syntactically correct (and therefore executable) ASM again. Together with some notion of partial equivalence or more precisely some notion of equivalence relative to a slicing criterion, we formally introduce the notion of slice.

The state resp. the behaviour of a program considered in [6] is defined by the values of its program variables. For ASMs, a state is defined by the content of its locations (see [4]). Locations can be described by atomic formulae.

**Definition 1 (Slicing Criterion)** A slicing criterion is a set of atomic formulae (possibly containing variables).

As in the classic case of program slicing, a slice of an ASM \( II \) results from \( II \) by removing parts from \( II \) and resulting in a syntactically correct ASM again. The formal definition proceeds via the notion of subprogram.

Therefore, we introduce the notions of *subformula* of a first-order formula and *subrule* of an ASM. Note that they do not quite correspond to the usual definitions of these terms. The main point is that in order to obtain subformulae or subrules, we remove a part of the formula or program. The removed pieces might consist of more than one part, the parts are not necessarily connected and they can also be situated somewhere in the middle of the formula or the program. The formal definition of these notions is displayed in figure 1 and figure 2 where sub(\( \psi \)) abbreviates the set of subformulae of a FO-formula \( \psi \) and sub(\( II \)) abbreviates the set of subrules of an ASM \( II \). A subrule without free variables is called a *subprogram*.

As a state of an ASM is defined by the contents of its locations, we want to be able to conclude from a state and a slicing criterion (and therefore from atomic formulae) to the corresponding locations.
**Definition 2** Let $A$ be a state and $S$ be a slicing criterion.

$L(A,S)$ denotes the set of all locations $l = (Q,a_1 \ldots a_r)$ in $A$ such that the atom $Qa_1 \ldots a_r$ is an instance of an atomic formula $\alpha \in S$.

We define the notion of $S$-equivalence on a state $A$ in order to be able to define $S$-equivalence in general where $S$ is a slicing criterion.

**Definition 3 (S-equivalence)** Let $\Pi$ and $\Pi'$ be ASMs and $S$ be a slicing criterion.

1. For a state $A$ and a set $L$ of locations of $A$, let $A|_L$ be the set of all pairs $(l,a)$ where $l \in L$ and $a$ is the content of $l$ in $A$.

2. Given a state $A$ of $\Pi$ resp. $\Pi'$. If for every run of $\Pi$ $A = A_0 \vdash_\Pi A_1 \vdash_\Pi$  
   $\ldots \vdash_\Pi A_i \vdash_\Pi A_{i+1} \vdash_\Pi \ldots$ there is a run of $\Pi'$ $A = A_0 \vdash_{\Pi'} A_1' \vdash_{\Pi'} \ldots \vdash_{\Pi'} A_i' \vdash_{\Pi'} A_{i+1}' \vdash_{\Pi'} \ldots$ such that $A_i|_{L(A_i,S)} = A_i'|_{L(A_i',S)}$ for all $i \in \mathbb{N}$ then $\Pi'$ $S$-captures $\Pi$ on $A$ ($\Pi \leq^S S \Pi'$).

3. Given a state $A$ of $\Pi$ resp. $\Pi'$. $\Pi$ and $\Pi'$ are called $S$-equivalent on $A$ ($\Pi \equiv^A S \Pi'$) if $\Pi \leq^A S \Pi'$ and $\Pi' \leq^A S \Pi$.

4. If $\Pi \equiv^A S \Pi'$ for all states $A$ of $\Pi$ resp. $\Pi'$ then $\Pi$ and $\Pi'$ are called $S$-equivalent ($\Pi \equiv_S S \Pi'$).

For deterministic ASMs, the notion of $S$-capturing and $S$-equivalence coincide.

Informally, one could formulate the condition for $\Pi \equiv_S S \Pi'$ as follows. $\Pi'$ behaves on all locations $(R,\bar{a})$ in exactly the same way as $\Pi$ where the atom $R\bar{a}$ is an instance of an atomic formula $\alpha \in S$.

**Definition 4 (Slice)** Let $\Pi$ be an ASM-program and $S$ be a slicing criterion. An $S$-slice of $\Pi$ is an ASM-program $\Pi_S \in \text{sub}(\Pi)$ such that $\Pi \equiv_S S \Pi_S$.
Remark. The set of locations, on which both programs must behave in the same way, depends on the state. It is respectively generated by the state and the slicing criterion. If import is not used then this set depends only on the initial state of a run.

In the rest of this work, we refer to the size of a formula as the number of symbols in the string representing it where variables, constant and relation symbols are counted as one. For a formula \( \varphi \), \(|\varphi|\) denotes its size.

The size of an ASM-rule is defined analogously but keywords are counted as one (then, endif, endforall, endchoose, enddo are not counted separately), and the size of the guards is defined by the above definition of the size of a formula. For an ASM \( \Pi \), \(|\Pi|\) denotes its size.

Note that in the following considerations, we could use any notion of size that is computable from an ASM or a formula.

**Definition 5 (Minimal Slice)** A minimal slice of an ASM-program \( \Pi \) and a slicing criterion \( S \) is an \( S \)-slice \( \Pi_S \) of \( \Pi \) such that there is no \( S \)-slice \( \tilde{\Pi} \) of \( \Pi \) with \(|\tilde{\Pi}| < |\Pi_S|\).

**3 Existence, Computability and Non-uniqueness**

The existence of a minimal \( S \)-slice (of an ASM-program \( \Pi \) for a slicing criterion \( S \)) is obvious as \( \text{sub}(\Pi) \) is finite for any ASM \( \Pi \) and \( \Pi \) itself is an \( S \)-slice for any slicing criterion \( S \). Note that the finiteness of \( \text{sub}(\Pi) \) does not imply that a minimal slice is computable. In general, it is not decidable whether an element of \( \text{sub}(\Pi) \) is a slice of \( \Pi \). This can be proven rather easily.

Let \( \varphi \) be a first-order formula not containing the nullary relation symbol Mode. Skip is a minimal \{Mode\}-slice of \( \Pi \) = if \( \varphi \) then Mode := true endif iff \( \varphi \) is not satisfiable. Furthermore, Skip is the only minimal \{Mode\}-slice of \( \Pi \) in this case. Consequently, the undecidability of the finite satisfiability problem for first-order logic FO induces the non-computability of a minimal slice for deterministic ASMs (not using import).

Furthermore, a minimal slice is not unique. Consider for example the ASM defined by do – in – parallel \( R_1 \ R_2 \) enddo where the programs \( R_1 \) and \( R_2 \) are defined as follows.

\[
\begin{align*}
R_1: & \quad \text{forall } xy : Rxy \text{ do} \\
& \quad \text{do-in-parallel} \\
& \quad \quad \text{if } ((P_1x \land P_2y) \lor P_3xy) \text{ then } Qxy := \text{true endif} \\
& \quad \quad \text{if } (P_2x \land P_1y) \text{ then } Qxy := \text{true endif} \\
& \quad \quad \text{enddo} \\
& \quad \text{endforall} \\
R_2: & \quad \text{forall } xy : Rxy \text{ do} \\
& \quad \text{do-in-parallel} \\
& \quad \quad \text{if } (P_1x \land P_2y) \text{ then } Qxy := \text{true endif} \\
& \quad \quad \text{if } ((P_2x \land P_1y) \lor P_3xy) \text{ then } Qxy := \text{true endif} \\
& \quad \quad \text{enddo} \\
& \quad \text{endforall}
\end{align*}
\]
$R_1$ and $R_2$ are equivalent and have the same size. Consequently, $II$ has two minimal $\{Qxy\}$-slices, namely $R_1$ and $R_2$ (which are different).

### 4 Guardedness Condition for ASMs

For obtaining positive computability results, we have to put restrictions on the ASMs. In order to benefit from the computability, the expressive power of the classes resulting from these restrictions should be rather high.

One class satisfying these conditions is obtained by taking only those ASMs that satisfy some guardedness condition. Informally, every first-order quantifier (possibly appearing in some guard) and every appearance of forall or choose is relativized by an atomic formula. We have mentioned the choice of an element because the following definition is not only applicable to deterministic ASMs but to any class of ASMs. We have chosen this general definition as we will later also consider nondeterministic ASMs.

In the main part of this work, we consider deterministic ASMs not using import.

**Definition 6** $D$ is the class of all deterministic ASMs not using import.

The definition of the guarded fragment $GF(C)$ of a class $C$ of ASMs uses the guarded fragment of first-order logic, that has originally been introduced in [1].

**Definition 7 (Guarded Fragment of First-Order Logic)** The guarded fragment ($GF$) of first-order logic is defined by induction as follows

- every relational atomic formula belongs to GF.
- $GF$ is closed under propositional connectives $\neg$, $\land$, $\lor$.
- if $\bar{x}$, $\bar{y}$ are tuples of variables, $\alpha(\bar{x}, \bar{y})$ is an atomic formula and $\varphi$ is a formula in GF such that $\text{free}(\varphi) \subseteq \text{free}(\alpha) = \{\bar{x}, \bar{y}\}$, then also the formulae $\exists \bar{y}(\alpha(\bar{x}, \bar{y}) \land \varphi(\bar{x}, \bar{y}))$ and $\forall \bar{y}(\alpha(\bar{x}, \bar{y}) \rightarrow \varphi(\bar{x}, \bar{y}))$ belong to $GF$.

In $GF$, all quantifications are guarded by atomic formulae. Note that formulae from $GF$ contain only relation and constant symbols. An important property of $GF$ is that in spite of its high expressive power, the (finite) satisfiability problem is decidable for $GF$ (see [1]).

**Definition 8** The guarded fragment $GF(C)$ of a class $C$ of ASMs is the subset of $C$ such that every element $II$ of $GF(C)$ uses only relation and constant symbols and satisfies the following conditions:

- the left-hand side and the right-hand side of any update rule is a boolean-valued term
- every guard of an if-clause in $II$ is in the guarded fragment of first-order logic
- every forall-rule has the form $\text{forall } \bar{x} : \alpha(\bar{x}, \bar{y}) \text{ } R \text{ endforall}$ where $\alpha(\bar{x}, \bar{y})$ is an atomic formula and $\text{free}(R) \subseteq \text{free}(\alpha) = \{\bar{x}, \bar{y}\}$
– every choice-rule (choice of an element from the domain) has the form
choose \( \bar{x} : \alpha(\bar{x}, \bar{y}) \) R endchoose where \( \alpha(\bar{x}, \bar{y}) \) is an atomic formula and
free(R) \( \subseteq \) free(\( \alpha \)) = \{\( \bar{x}, \bar{y} \}\)

In the next two sections, we prove the main proposition of this work.

**Theorem 9** There exists an algorithm computing for a slicing criterion \( S \) and an ASM \( \Pi \in \text{GF}(\mathcal{D}) \) a minimal \( S \)-slice of \( \Pi \).

In the rest of this section, we consider the expressive power of \( \text{GF}(\mathcal{D}) \).

\( \text{GF}(\mathcal{D}) \) has a rather high expressive power. In order to demonstrate this, we compare the expressive power of \( \text{GF}(\mathcal{D}) \) to the expressive power of other well-known and expressive mechanisms.

The first such mechanism is Datalog LITE (see [3]). This is a guarded version of stratified Datalog additionally allowing a limited form of universal quantification in rule bodies.

**Proposition 10** \( \text{GF}(\mathcal{D}) \) is strictly more expressive than Datalog LITE.

Well-known logical formalisms such as propositional multi-modal logic, computation tree logic (CTL), the alternation free \( \mu \)-calculus, each correspond to well-defined and syntactically simple fragments of Datalog LITE.

Furthermore, it has been proven in [3] that Datalog LITE is equivalent in expressive power to alternation-free guarded fixed point logic, a natural fragment of the guarded fixed point logic obtained by disallowing alternations between least and greatest fixed points. Guarded fixed point logic is the extension of GF by least and greatest fixed points. For \( \text{GF}(\mathcal{D}) \), we can skip the restriction of alternation-freedom necessary for the result on Datalog LITE and even obtain a strict capturing result.

**Proposition 11** \( \text{GF}(\mathcal{D}) \) is strictly more expressive than guarded (first-order) fixed point logic.

The intuitive reason why we can abandon the restriction of alternation-freedom is the following. A Datalog LITE program is arranged in strata which can only be processed in a top-down order. Once a stratum has been left, it is not possible to return to it again. This is the main reason for the necessity of alternation freedom. For \( \text{GF}(\mathcal{D}) \), we do not have any restriction on the order of executing parts of an ASM and we can abandon alternation-freedom.

The above facts show us the expressive power of \( \text{GF}(\mathcal{D}) \) but nevertheless, we might still miss an intuition about this class. In order to get a basic understanding of it, we consider the special case of graphs as states. I.e., the vocabulary contains only the binary relation symbol \( E \).

Let \( \Pi \in \text{GF}(\mathcal{D}) \) with vocabulary \( \{E/2\} \) and \( \mathcal{A} = (V, E) \) be a state of \( \Pi \). Then for all states \( \mathcal{B} = (V, E') \) of \( \Pi \) with \( \mathcal{A} \models^* \Pi \mathcal{B} \), \( E' \subseteq E \). The reason is that we can only change the interpretation of \( E \) on pairs \((x, y)\) that are connected via
some edge. As soon as $x$ and $y$ are no more connected, we have no possibility to access this pair.

These considerations can be generalized to arbitrary ASMs from $\text{GF}(\mathcal{D})$ over relational vocabulary with relations of arbitrary arity and tuples of arbitrary length (bounded by the maximal arity of the relation symbols in the vocabulary). In this case, the respective components of a tuple have to coincide in a tuple contained in some relation.

As a consequence of the above observation, we get some non-computability results. E.g., the transitive closure is not computable by an ASM from $\text{GF}(\mathcal{D})$.

The above considerations demonstrate the high expressive power of $\text{GF}(\mathcal{D})$. We defer a detailed investigation of the computational power of $\text{GF}(\mathcal{D})$ to another paper.

5 Example

In this section, we exemplarily consider the ASM $\Pi \in \text{GF}(\mathcal{D})$ (see figure 3). Essentially, an input of $\Pi$ is a transition system with a set of distinguished nodes. The intended interpretations of the relations used in $\Pi$ are the following.

For two states $x$ and $y$ of the transition system $\mathcal{T}$ and an action $a$, $\text{Taxy}$ holds iff a transition from $x$ to $y$ using $a$ exists. $P$ contains exactly the safe states of $\mathcal{T}$. $S$ contains exactly the distinguished (starting) states of $\mathcal{T}$.

Initially, $R$ is intended to be empty, liveness is equal to false and safe is equal to true. When the ASM halts (i.e., no more changes in the interpretations happen), $R$ contains those states of $\mathcal{T}$ that are reachable from the set $S$ via a finite sequence of actions. Furthermore, safe is true iff all reachable states are safe and liveness is true iff every reachable node has an outgoing edge.

It is not necessary to take into account the complete program in order to observe the evaluation of only a part of the relations. This observation is reflected by its slices. Assume that $\Pi$ is part of an ASM $\tilde{\Pi}$ (but the relations in $\Pi$ are not updated outside $\Pi$). A wrong evaluation of $R$ might be the reason for an error observed somewhere else in $\tilde{\Pi}$. Therefore, we are possibly interested only in that part of $\tilde{\Pi}$ that influences the evaluation of $R$. Consequently, it suffices to consider an $\{Rx\}$-slice of $\Pi$. There exists exactly one minimal $\{Rx\}$-slice (see figure 3). Though $R$ influences the evaluation of safe and liveness, the other direction is not true. The evaluation of $R$ does not depend on these relations. $R$ depends only on the static unary relation $S$. Therefore, we can cut off all updates of other relations in order to obtain an $\{Rx\}$-slice of $\Pi$. Furthermore, it is not necessary to embed the update rule $Ry := \text{true}$ in a conditional rule with guard $\neg Ry$. The rule $\text{if } \neg Ry \text{ then } Ry := \text{true endif}$ is equivalent to the rule $Ry := \text{true}$. However, it might be intuitive to embed the update rule in a conditional rule as the content of a location $(R, a)$ is only changed if it is false in the current state.

If we are only interested in the evaluation of safe, we can restrict our considerations to a $\{\text{safe}\}$-slice of $\Pi$ and there is exactly one minimal $\{\text{safe}\}$-slice.
do-in-parallel
forall x : Sx do
  Rx := true
endforall
forall x : Rx do
  forall ay : Taxy do
    if \neg Ry then Ry := true endif
  endforall
endforall
forall x : Rx do
  do-in-parallel
    if \neg \exists ay Taxy then lifeness := false endif
    if \neg Px then safe := false endif
  enddo
endforall
enddo

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\text{do-in-parallel} & \text{enddo} \\
\hline
forall x : Sx do & forall x : Sx do \\
  Rx := true & Rx := true \\
endforall & endforall \\
forall x : Rx do & forall x : Rx do \\
  forall ay : Taxy do & forall ay : Taxy do \\
    if \neg Ry then Ry := true endif & Ry := true endif \\
  endforall & endforall \\
endforall & endforall \\
\hline
\end{tabular}
\caption{ASM $\Pi$ (left) and its minimal $\{Rx\}$-slice $\Pi_R$ (right)}
\end{figure}

\begin{align*}
do-in-parallel \\
\bar{\Pi}_R \\
forall x : Rx do \\
  \text{if } \neg Px \text{ then } \text{safe} := \text{false} \text{ endif} \\
endforall \\
enddo
\end{align*}

where $\bar{\Pi}_R$ results from $\Pi_R$ by removing \texttt{do-in-parallel} and \texttt{enddo}.

From this slice, we see that a slice may involve updates of locations whose lefthand side does not fit to an element of the slicing criterion. In this special case, the $\{\text{safe}\}$-slice contains updates of the relation $R$. It is clear that this can not be avoided as the computation of safe strongly uses the relation $R$ in a non-trivial way. Though safe and lifeness are computed in parallel, the evaluation of safe does not involve lifeness. Consequently, we can cut off all updates of lifeness in order to obtain a $\{\text{safe}\}$-slice of $\Pi$.

Note that the minimal $\{\text{safe}\}$-slice is equal to the minimal $\{\text{safe}, Rx\}$-slice.

\section{Quasistates and Partial Equivalence}

The main goal of this paper is an algorithm computing for a slicing criterion $S$ and an ASM $\Pi \in \text{GF}(D)$ a minimal $S$-slice of $\Pi$. This result uses the notion of quasistate. Essentially, every quasistate represents a class of states that evaluate certain formulae to the same values. The same behaviour on all runs resp. $S$-equivalence is reduced to the same behaviour on the runs of the standard states of the quasistates. In the proof of the computability result, the formulae occurring in the ASM are evaluated on quasistates. The idea of the notion of quasistate is similar to the one of quasimodel introduced in [1].
6.1 Quasistates

For a finite set $F$ of first-order formulae (FO-formulae), an $F$-type is a maximal consistent subset of $F$.

For a sequence of variables $\bar{y}$, two types $\Delta$ and $\Delta'$ are called $\bar{y}$-close ($\Delta =_{\bar{y}} \Delta'$) if $\Delta$ and $\Delta'$ have the same formulae with free variables disjoint from $\bar{y}$.

**Definition 12 (Quasistate)** Let $F$ be a finite set of FO-formulae.

An $F$-quasistate is a set of $F$-types $T$ such that for each $\Delta \in T$ and each formula $\exists \bar{y} \varphi \in \Delta$ there is a type $\Delta' \in T$ with $\varphi \in \Delta'$ and $\Delta =_{\bar{y}} \Delta'$.

$\Phi$ holds in a quasistate if $\Phi \in \Delta$ for some $\Delta$ in this quasistate.

In particular, the $\bar{y}$-closeness implies for every sentence $\varphi \in F$ that $\varphi \in \Delta$ for all $\Delta \in Q$ or $\varphi \notin \Delta$ for all $\Delta \in Q$.

For an $F$-quasistate $Q$, we define a standard state $A_Q$. The idea of this construction is that $A_Q$ realizes exactly the quasistate $Q$.

$\pi$ is a path $\pi = \langle \Delta_1, \Phi_1, \ldots, \Delta_n, \Phi_n \rangle$ if $\Delta_1$ and $\Delta_{n+1}$ are types in $Q$, each formula $\Phi_i$ is of the form $\exists \bar{y} \varphi \in \Delta_i$, $\varphi \in \Delta_{i+1}$ and $\Delta_{i+1} =_{\bar{y}} \Delta_i$. We say that the variables in $\bar{y}$ changed their values from $\Delta_i$ to $\Delta_{i+1}$ whereas the others did not. Furthermore, a variable $z$ is called new in a path $\pi$ if either $|\pi| = 1$ or the value of $z$ was changed in the last round in $\pi$.

Now, we define $A_Q$. The domain of $A_Q$ consists of all pairs $(\pi, z)$ where $\pi$ is a path and $z$ is new in $\pi$. In the rest of this paragraph, we give the interpretation of the predicates. For a relation $R$, $I(R)$ holds of the sequence of elements $\langle (\pi_j, x_j) \rangle_{j \in J}$ iff the paths $\pi_j$ fit into one linear sequence under inclusion, with maximal path $\pi^*$ such that $\Delta^*$ (the last type of $\pi^*$) contains $R(\langle x_j \rangle_{j \in J}$ and $x_j$ does not change its value on the further path to the end of $\pi^*$ for any $(\pi_j, x_j)$.

In the preceding part of this subsection, we have explained how to construct for a quasistate $Q$ a state $A_Q$ realizing $Q$. Now, we consider the other direction. I.e., for a state $A$, we give the maximal $F$-quasistate $Q^F_A$ that is realized by $A$. For an $F$-quasistate $Q$ we say that the state $A$ realizes $Q$ if $A$ realizes every $\Delta \in Q$.

**Definition 13** Let $F$ be a finite set of FO-formulae and $A$ be a state over the same vocabulary. The $F$-quasistate of $A$ is defined as:

$$Q^F_A := \{ \Delta : \Delta \text{ is an } F \text{-type that is realized in } A \}$$

6.2 $S$-Equivalence on Quasistates

In this subsection, we prepare the proof of our main result. First, we define for an ASM $\Pi$ the set $T_\Pi$ of all formulae whose valuation possibly influences the run of an ASM $\Pi$.

**Definition 14** 1. For an ASM $\Pi$, $T_\Pi$ is the set of all FO-formulae over the vocabulary of $\Pi$ such that their size is $\leq |\Pi|$ and only variables from $\Pi$ are used.

2. For ASMs $\Pi_1$ and $\Pi_2$, $T_{\Pi_1, \Pi_2}$ denotes the set $T_{\Pi_1} \cup T_{\Pi_2}$.
The $T_{\Pi_1,\Pi_2}$-quasisate of a state $A$ reflects the valuation of those formulae in $A$ that may influence a run of $\Pi_1$ or $\Pi_2$ on $A$. Therefore, we can restrict to the valuations of these formulae in order to reason about runs of $\Pi_1$ and $\Pi_2$. All other formulae do not have any influence.

In the rest of this work, $Q_{\Pi_1,\Pi_2}^A$ abbreviates $Q_{\Pi_1,\Pi_2}^{T_{\Pi_1,\Pi_2}}$ and $Q_A^\Pi$ abbreviates $Q_{T_{\Pi_1,\Pi_2}}^A$.

In the previous subsection, we have defined the $F$-quasisate of a state $A$. Analogously, one can define the $F$-type of a tuple in a state $A$.

Let $F$ a finite set of FO-formulae, $A$ be a state and $V = \{x_1, \ldots, x_n\}$ be the set of variables of $F$. Any tuple $\bar{a} = a_1 \ldots a_n$ of elements from the domain $A$ of $A$ induces a variable assignment $\sigma_{\bar{a}} : V \rightarrow A$ such that $\sigma_{\bar{a}}(x_i) = a_i$ for $i \in \{1, \ldots, n\}$ (the variable assignment induced by $\bar{a}$).

The $T_{\Pi_1,\Pi_2}$-type $\Delta_{\Pi_1,\Pi_2}^{\bar{a}}$ of $\bar{a}$ in $A$ is the $T_{\Pi_1,\Pi_2}$-type $\Delta \in Q_{\Pi_1,\Pi_2}^A$ that contains exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ with $A, \sigma_{\bar{a}} \models \varphi$. Note that $\Delta_{\Pi_1,\Pi_2}^{\bar{a}}$ is unique. For an ASM $\Pi$, $\Delta_{\Pi_1,\Pi_2}^{\bar{a}}$ abbreviates $\Delta_{\Pi_1,\Pi_2}^{\Pi,\bar{a}}$.

Since $T_{\Pi}$ contains all formulae whose valuation influences a computation step of $\Pi$, the following lemma holds.

**Lemma 15** Let $A_1$ and $A_2$ be states of an ASM $\Pi \in \mathcal{D}$, $\bar{a}$ a tuple of elements from the domain of $A_1$, $\bar{b}$ a tuple of elements from the domain of $A_2$ such that $\Delta_{A_1,\bar{a}}^{\Pi} = \Delta_{A_2,\bar{b}}^{\Pi}$. Furthermore, let $A_1 \vdash_\Pi B_1$ and $A_2 \vdash_\Pi B_2$. Then $\Delta_{B_1,\bar{a}}^{\Pi} = \Delta_{B_2,\bar{b}}^{\Pi}$.

By induction on the number of steps, we obtain the following corollary.

**Corollary 16** Let $A$ and $B$ be states of an ASM $\Pi \in \mathcal{D}$, $\bar{a}$ a tuple of elements from the domain of $A$, $\bar{b}$ a tuple of elements from the domain of $B$ such that $\Delta_{A,\bar{a}}^{\Pi} = \Delta_{B,\bar{b}}^{\Pi}$. Furthermore, let $A = A_0 \vdash_\Pi A_1 \vdash_\Pi A_2 \vdash_\Pi \cdots \vdash_\Pi A_i \vdash_\Pi \cdots$ resp. $B = B_0 \vdash_\Pi B_1 \vdash_\Pi B_2 \vdash_\Pi \cdots \vdash_\Pi B_i \vdash_\Pi \cdots$ the run of $\Pi$ on $A$ resp. $B$. Then, for all $i \in \mathbb{N}$ $\Delta_{A_i,\bar{a}}^{\Pi} = \Delta_{B_i,\bar{b}}^{\Pi}$.

The following lemma provides the possibility to conclude from standard states to arbitrary states.

**Lemma 17** Let $\Pi_1, \Pi_2 \in \mathcal{D}$ and $Q$ be a $T_{\Pi_1,\Pi_2}$-quasisate. Furthermore, let $A_Q$ be the standard state of $Q$ and $S$ be a slicing criterion. If $\Pi_1 \equiv_{S}^A \Pi_2$ then $\Pi_1 \equiv_{T}^{\Delta} \Pi_2$ for all states $A$ with $Q_A^{\Pi_1,\Pi_2} = Q$.

**Proof.** Let $\Pi_1, \Pi_2 \in \mathcal{D}$ and $Q$ be a $T_{\Pi_1,\Pi_2}$-quasisate. Furthermore, let $A_Q$ be the standard state of $Q$ and $S$ be a slicing criterion. W.l.o.g., we can assume that $S \subseteq T_{\Pi_1,\Pi_2}$ as locations $(R, \bar{a})$ where $R\bar{a}$ is not an instance of an element of $T_{\Pi_1,\Pi_2}$ are static for $\Pi_1$ and for $\Pi_2$. Consequently, $\Pi_1$ and $\Pi_2$ behave on these locations in the same way.

Let $A$ be any state whose $T_{\Pi_1,\Pi_2}$-quasisate is $Q$ and let $A = A_0 \vdash_{\Pi_1} A_1 \vdash_{\Pi_1} A_2 \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} A_i \vdash_{\Pi_1} \cdots$ resp. $A = B_0 \vdash_{\Pi_2} B_1 \vdash_{\Pi_2} B_2 \vdash_{\Pi_2} \cdots \vdash_{\Pi_2} B_i \vdash_{\Pi_2} \cdots$ be the run of $\Pi_1$ resp. $\Pi_2$ on $A$.  

For any $n$-tuple $\bar{a}$ (where $n$ is the number of variables occurring in $T_{\Pi_1,\Pi_2}$) of elements from the domain of $\mathcal{A}$, there is a type $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\bar{a}} \in Q$ that contains exactly those formulae $\varphi \in T_{\Pi_1,\Pi_2}$ with $\mathcal{A}, \sigma_{\bar{a}} \models \varphi$ ($\sigma_{\bar{a}}$ is the variable assignment induced by $\bar{a}$, see above). According to the definition of $\mathcal{A}_Q$, there is an $n$-tuple $\bar{b}$ of elements from the domain of $\mathcal{A}_Q$ (the standard state for $Q$) such that $\mathcal{A}_Q, \sigma_{\bar{b}} \models \varphi$ holds for all such formulae $\varphi \in T_{\Pi_1,\Pi_2}$ that are contained in $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\bar{a}}$. Consequently, $\bar{b}$ realizes $\Delta^{\Pi_1,\Pi_2}_{\mathcal{A},\bar{a}}$ in $\mathcal{A}_Q$.

Using corollary 16, we obtain that $\Delta^{\Pi_1,\bar{a}} = \Delta^{\Pi_1,\bar{b}}$ and $\Delta^{\Pi_2,\bar{a}} = \Delta^{\Pi_2,\bar{b}}$ for all $i \in \mathbb{N}$. Assuming that $\Pi_1 \equiv^A_S \Pi_2$, this implies that $\Delta^{\Pi_1,\bar{a}} = \Delta^{\Pi_2,\bar{a}}$ for all $i \in \mathbb{N}$.

As $\bar{a}$ has been chosen arbitrarily, the above argumentation holds for any $n$-tuple of elements from the domain of $\mathcal{A}$. Consequently, we obtain that for all $i \in \mathbb{N}$ $\mathcal{A}_i|_{L(\mathcal{A}_i,S)} = B_i|_{L(B_i,S)}$. By definition, this is equivalent to $\Pi_1 \equiv^A_S \Pi_2$. □

For ASMs $\Pi_1, \Pi_2$ from $\text{GF}(D)$, we can restrict our considerations to the subset $T_{\Pi_1,\Pi_2} \cap \text{GF}$ of $T_{\Pi_1,\Pi_2}$. In [1], it has been proven that a GF-formula $\Phi$ is satisfied by some state iff $\Phi$ holds in some quasistate. Using this fact and lemma 17, we obtain the following lemma.

**Lemma 18** Let $\Pi_1, \Pi_2 \in \text{GF}(D)$. If $\Pi_1 \equiv^A_S \Pi_2$ holds for all $(T_{\Pi_1,\Pi_2} \cap \text{GF})$-quasistates $Q$ then $\Pi_1 \equiv_S \Pi_2$.

Note that this is the point where we need to introduce the restriction to the guarded fragment. The reason is that the proposition from [1] does not hold for arbitrary FO-formulae but for formulae from GF.

### 7 Computing a Minimal Slice

In this section, we give the proof of theorem 9 stating that there exists an algorithm computing for a slicing criterion $S$ and an ASM $\Pi \in \text{GF}(D)$ a minimal $S$-slice of $\Pi$.

**Proof of Theorem 9.** From lemma 18 we know the following. In order to check for two ASMs $\Pi_1$ and $\Pi_2$ from $\text{GF}(D)$, whether $\Pi_1 \equiv_S \Pi_2$, it is sufficient to check whether $\Pi_1 \equiv^A_S \Pi_2$ for all $T_{\Pi_1,\Pi_2}$-quasistates $Q$.

For a $T_{\Pi_1,\Pi_2}$-quasistate $Q$, one can check whether $\Pi_1 \equiv^A_S \Pi_2$ as follows.

As the domain $\mathcal{A}_Q$ of $\mathcal{A}_Q$ is finite, there is only a finite number $a_q$ of reachable states over $\mathcal{A}_Q$. $a_q$ is computable from the vocabulary of $\mathcal{A}_Q$ and the size of its domain. More precisely, we can assume $a_Q$ to be $\prod_{R/n_R \in \mathcal{Y}}$ is a relation symbol $2^{|\mathcal{A}_Q|^n}$ where $\mathcal{A}_Q$ is the domain of $\mathcal{A}_Q$ and $\mathcal{Y}$ is the vocabulary of $\Pi_1$ resp. $\Pi_2$. Note that constant symbols do not need to be taken into account as their interpretation can not be changed by an ASM from $\mathcal{D}$.

To check whether $\Pi_1 \equiv^A_S \Pi_2$, construct the run of $\Pi_1$ up to the length $a_q + 1$ and the respective initial segment of the run of $\Pi_2$. This leads to $\mathcal{A}_Q = \mathcal{A}_0 \vdash_{\Pi_1} \mathcal{A}_1 \vdash_{\Pi_1} \mathcal{A}_2 \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_i \vdash_{\Pi_1} \cdots \vdash_{\Pi_1} \mathcal{A}_{a_Q+1}$ and $\mathcal{A} = \mathcal{B}_0 \vdash_{\Pi_2}$.
\[ \mathcal{B}_1 \vdash \mathcal{B}_2 \vdash \mathcal{B}_3 \vdash \cdots \vdash \mathcal{B}_i \vdash \mathcal{A}_i \mathcal{B}^Q_{aQ+1}. \] Now, check whether \( \mathcal{A}_i|_{L(A, S)} = \mathcal{B}_i|_{L(A', S)} \) for all \( i \in \{1, \ldots, aQ + 1\} \).

For two ASMs \( \mathcal{A}_1 \) and \( \mathcal{A}_2 \), the set of \( (T_{\mathcal{A}_1, \mathcal{A}_2} \cap GF) \)-quasistates is finite. Therefore, it is decidable whether \( \mathcal{A}_1 \equiv_S \mathcal{A}_2 \) for \( \mathcal{A}_1, \mathcal{A}_2 \in GF(D) \). This is a consequence of lemma 18.

As \( \text{sub}(\mathcal{A}_1) \) is finite for every ASM \( \mathcal{A}_1 \) and computable from \( \mathcal{A}_1 \), we are now able to formulate the following algorithm computing for a slicing criterion \( S \) and an ASM \( \mathcal{A}_1 \) a minimal \( S \)-slice of \( \mathcal{A}_1 \).

For every program \( \mathcal{A}_i \in \text{sub}(\mathcal{A}_1) \), check whether \( \mathcal{I}_1 \equiv_S \mathcal{A}_i \). Return a minimal ASM from \( \text{sub}(\mathcal{A}_1) \) satisfying this condition. \( \square \)

8 Complexity

From the proof of theorem 9 and the proofs of the involved lemmas, we can directly derive an algorithm computing a minimal slice. There exists a polynomial \( p \) such that this algorithm outputs a minimal after at most \( 2^{2p(|\mathcal{A}_1|)} \) steps for any ASM \( \mathcal{A}_1 \in GF(D) \). Furthermore, we can easily reduce the finite satisfiability problem for GF to computing a minimal slice for ASMs from GF(D). This problem is 2EXPTIME-complete.

But note that the above complexities are those for the worst-case. For practical applications we would be more interested in the average-case complexity which is possibly much lower and even acceptable for practical purposes. But this one depends strongly on the applications and may differ for different ones such that it is not possible to give a uniform average-case complexity.

Moreover, the runtime of the algorithm in practice depends strongly on the concrete implementation of the algorithm. There is a number of optimizations that do not change the runtime in the worst-case but improve it in many cases. We give some examples in the following. First of all, one could try all subprogram increasingly ordered by size. Furthermore, the test of a subprogram can be breaked off as soon as there is a difference in a run (relative to the slicing criterion) to the respective run of the original program. While constructing the run of the subprogram, we can directly check this. There are many programs and subprograms where such a difference arises rather early. A third possibility of optimization arises from the observation that the size of \( T_{\mathcal{A}_1} \) strongly increases with the number of variables used in the program \( \mathcal{A}_1 \). But if e.g. two distinct variables are bounded by independent forall-rules then they could be substituted by the same variable without changing the semantics of the original ASM. This is e.g. the case if one has a program of the form \textbf{do – in – parallel} \( \Pi_1 \Pi_2 \textbf{enddo} \) where \( \Pi_1 \) and \( \Pi_2 \) are forall-rules binding distinct variables. Consequently, one could optimize an implementation of the algorithm by preprocessing the input program such that its semantics is not changed but the number of distinct variables is lowered.
9 Extensions of the Basic Result

Theorem 9 can be extended rather easily in different ways. We give two examples.

Extension of $\mathcal{D}$ by Nondeterminism. By additionally allowing the choice of a tuple of elements from the domain of a state, we can extend the class $\mathcal{D}$ of ASMs to the class $\mathcal{ND}$.

The proof of the computability of a minimal slice for nondeterministic ASMs from $GF(\mathcal{ND})$ can be done rather analogously to the one of theorem 9. The only difference is that instead of runs we have to consider computation graphs where a state may have more than one successor. In this extended scenario, the same argumentation as for $GF(\mathcal{D})$ can be used except that we do not reason about the successor of a state but about one successor and the existence of another one in the run of a subprogram with certain properties.

Extension of $\mathcal{D}$ by Import. By additionally allowing the use of import, we can extend the class $\mathcal{D}$ of ASMs to the class $\mathcal{DL}$.

W.l.o.g., we can assume that $v$ appears only on the left-hand side of an update rule in a rule $R$ appearing in the form $\text{import } v \text{ R endimport}$.

The proof of the computability of a minimal slice for deterministic ASMs from $GF(\mathcal{DL})$ proceeds similar to the one for deterministic ASMs from $GF(\mathcal{D})$.

As in the case without import we consider the runs on the standard states, but here, we modify the obtained structure after every step such that we do not get the real runs but reductions of them. The reason for the need of such a reduction is that the active domain of the states grows and therefore the argument that there are only finitely many states over the domain of the initial state does not work anymore. After each computation step, the set of imported elements is reduced to a representative set of limited size.

10 Variations in the Notion of Slicing

Dynamic and Static Slicing. In the literature, there are two basic versions of slicing, namely static and dynamic slicing. For dynamic slicing of a program, a specific input is given and we are only interested in a subprogram such that the behaviour is the same for this specific input. In static slicing, the input is not given. We are interested in the same behaviour for all possible inputs. Therefore, slicing in this work corresponds to static slicing.

For the classes $\mathcal{D}$ and $\mathcal{ND}$ (even if we allow function symbols to be in the vocabulary of an ASM in $\mathcal{D}$), dynamic slicing is rather easy. The computation graph of an ASM from $\mathcal{D}$ resp. $\mathcal{ND}$ is finite for a given initial state. Therefore, we can directly compare the computation graphs of two ASMs on a given state relative to a slicing criterion. Consequently, comparing the computation graph of an ASM $II$ and a given structure to the computation graphs of all elements of sub($II$) on this state leads to a minimal (dynamic) slice.
If the use of \texttt{import} is allowed then we can also compute a minimal (dynamic) slice assumed that we have only relation and constant symbols. The algorithm is similar to the one in the case without import but we do not consider the complete computation graph. We do not insert all states into the computation graph. If the difference between two states is only caused by importing different elements, then these states are identified and only one of them is inserted into the computation graph as node.

**Limiting the Number of Steps.** The idea of limiting the number of steps is that possibly we are not interested in the complete runs of an ASM but only in the initial segments of length $n \in \mathbb{N}$ of all runs. This applies for example if we know that an error occurs during the first $n$ steps of a computation.

Similar to the notion of $S$-slice, we can introduce the notion of $(S, n)$-slice which is a program from $\text{sub}(\Pi)$ that, relative to $S$, behaves in the same way as $\Pi$ on every initial segment of length $n$.

The proof of the computability of a minimal $(S, n)$-slice for $\text{GF}(\mathcal{D})$ can be done completely analogously to the proof of theorem 9 except that we have to consider the runs on the standard states $\mathcal{A}_Q$ only up to the length $n$.

**Conditioned Slicing.** The idea of conditioned slicing is that we consider only states satisfying a given property. Formally, this can be formulated as follows.

Similar to the notion of $S$-slice, we can introduce the notion of $(S, \varphi)$-slice which is a program from $\text{sub}(\Pi)$ that, relative to $S$, behaves in the same way as $\Pi$ on every run whose initial state satisfies $\varphi$.

The proof of the computability of a minimal conditioned slice for ASMs from $\text{GF}(\mathcal{D})$ and properties from $\text{GF}$ can be done completely analogously to the proof of theorem 9 except that we have to restrict to the quasistates satisfying $\varphi$.

**Semantic Slicing.** Instead of giving a slicing criterion as a set of atomic formulæ, one could also consider an ASM $\Pi$ and a property $\varphi$ (given some formalism resp. logic) such that one is interested in a minimal subprogram $\Pi'$ of $\Pi$ that behaves exactly the same way as $\Pi$ relative to $\varphi$.

Similar to the notion of $S$-slice, we can introduce the notion of $\varphi$-slice which is an element of $\text{sub}(\Pi)$ that, relative to the valuation of $\varphi$, behaves in the same way as $\Pi$ on every run.

The proof of the computability of a minimal semantic slice for ASMs from $\text{GF}(\mathcal{D})$ and properties from $\text{GF}$ can again be done similar to the one above with the following changes.

11 Conclusion and Future Work

We have presented an approach to transfer the notion of program slicing to ASMs. Although a minimal (static) slice is not computable in the general case, we have introduced an expressive class of ASMs for which a minimal (static)
slice is computable. This basic computability result can be extended to larger classes of ASMs. Furthermore, we have demonstrated how it can be transferred to further notions of slicing to ASMs. These continuative considerations can be done rather easily once we have obtained the basic notion of static slicing for ASMs and the respective proof of the computability result.

Currently, we are implementing a system providing the possibility to slice ASMs from the presented class. This implementation is based on the algorithm derivable from the computability proof. There is a number of possibilities for optimizing the algorithm as we have already explained in the section on the complexity. We are aiming to obtain a variant which computes minimal slices within acceptable time for most cases.

Acknowledgements

I want to thank Erich Grädel for many helpful remarks on earlier drafts of this paper. I also want to thank the anonymous referees for their useful suggestions.

References

The Cryptographic Abstract Machine

Dean Rosenzweig and Davor Runje

University of Zagreb

Abstract. The Cryptographic Abstract Machine is an executional model of cryptographic actions, independent of the concrete cryptographic procedures employed, even of the abstraction level of the underlying model of cryptography. This is motivated both by a theoretical purpose of relating the dynamics of protocol executions at different levels of abstraction, and by a practical purpose of enabling automatic generation of provably correct code implementing protocol roles from high level specifications. Here we define the CrAM and show how slightly refurbished message patterns of [RRS03] can be compiled to CrAM code both for analysis and for creation of messages, and prove the correctness and completeness of that compilation.

Introduction

The Cryptographic Abstract Machine (CrAM in the sequel) is an executional model of cryptographic actions, independent of the concrete cryptographic procedures employed, even of the abstraction level of the underlying model of cryptography. Three such levels can be discerned:

- abstract or formal model, where only the abstract structure of cryptographic messages is represented, typically by terms of a vocabulary, abstracting away from their concrete representation and concrete cryptographic algorithms, i.e. [Low96,FHG99,CDL+00,DMP03,BR97,BR98]...
- computational model, admitting that messages are bitstrings, operating under complexity-theoretic assumptions on cryptographic algorithms, see for instance [BN00,BDPR98] for splendid examples of the definitional effort;
- concrete programming APIs, relying on concrete message-encoding schemes and services of a concrete cryptographic library and/or device.

The last two levels are usually related by complexity-theoretic conjectures on combinatorial problems underlying cryptographic algorithms, such as factoring or discrete logarithm — for work towards relating the first two levels see [AR02,MW03].

We propose the CrAM in the context of a broader program, with the goal of deriving provably secure code implementing cryptographic protocols. We see the problem of achieving that goal decomposed as follows:

1. develop a language of abstract message patterns, akin to the usual messages-and-arrows representation, supporting formal proofs under the assumptions of abstract cryptography;

© Springer-Verlag Berlin Heidelberg 2004
2. decouple the message patterns from the assumptions of abstract cryptography, allowing their direct interpretation also in computational models and/or concrete implementations of cryptographic algorithms;

3. provide a framework for translating formal proofs into computational settings, under the common assumptions on cryptographic algorithms;

4. generate provably correct implementations of pattern-based protocol descriptions for a given programming language and a cryptographic programming library.

Aim 1 was largely accomplished in [RRS03]: the message pattern language is proven to be universal for analysis and synthesis of cryptographic messages under a model of abstract cryptography. A formal proof establishing impossibility of an attack with message patterns also proves impossibility of an attack in the context of abstract cryptography.

Aim 3 is left for future work. Our intent is to develop a framework for systematic translation of proofs for the abstract cryptography model into proofs for different computational models, with target theorems of form: abstractly safe protocol + computationally safe algorithms = computationally safe protocol, for different notions of ‘computationally safe’. It seems that the properties of abstract cryptography used in a formal proof, together with the computational security criterion desired, will largely dictate the computational security assumptions on algorithms needed. We see this future work as direct continuation and application of [AR02,MW03] for analysis of static messages and [MW03,War03] for dynamic protocol execution. The CrAM seems to be the right setting, since it provides a simple and precise way of saying that cryptographic agents at different abstraction levels ‘do the same’: they execute the same CrAM program in different environments.

This paper intends to realize most (though not all) of aims 2 and 4.

We start, in section 1, by laying down the vocabulary and the assumptions common to all abstraction levels we rely on throughout the paper.

In section 2 we revisit the message pattern language of [RRS03] in order to allow different models of cryptography, at different levels of abstraction, to be plugged into patterns. For the purpose of this paper, we intentionally disregard interpretations of cryptographic objects, and place a minimal set of assumptions on a model of cryptography needed to prove equivalence of message patterns and CrAM programs. The modified language of message patterns recaptures the universality properties of patterns wrt the model of abstract cryptography of [RRS03].

The main result of this paper is generation of provably correct implementation of a pattern-based protocol description for a given programming language and a cryptographic programming library. This is accomplished by compiling patterns to CrAM code, in a provably correct and complete way. In view of the representation of protocol roles of [RRS03] as sequences of match–create pairs of patterns, we obtain compilation of protocol roles to CrAM code automatically, by pasting and gluing with instructions for input–output.
The intuition of a protocol role as an interactive machine with input and output of messages gets a concrete form in the CrAM, which is now also decoupled from any specific model of encryption, and works with any reasonable model supported by the vocabulary and assumptions.

We show that both compilation algorithms are both correct and complete, which means that CrAM programs are at least as strong as patterns, in any specific incarnation of both.

In section 3 we define the CrAM, and in section 4 we define the compilation. Proofs of security for pattern–based protocol descriptions, to be done in the completion of aim 3 indicated above, will compose with our correctness and completeness proofs presented here so as to prove security of CrAM–implementations of protocols. A CrAM–implementation becomes a real implementation either by interpreting the CrAM directly (one way of doing that is AsmL code to be found at [Web]), or by expanding CrAM programs into programs of the target implementation language in a provably correct way. The latter possibility is the only really practical one, but conceptually it is *deja vu*, and is skipped here to conserve some space. In view of our technical framework, that of ASMs as implemented in AsmL, it is rather straightforward to expand CrAM code to provably correct code in any language with a well defined ASM semantics, such as C, C++, Java [GH93,HS00,Wal95,SSB01].

The results presented here are in a sense independent of the ASM framework. The justification for relying on ASMs and AsmL (if one is still needed) is threefold:

- The framework supports our model of abstract cryptography [RRS03] well.
- Main results of this paper are essentially on compiler correctness/completeness, and ASMs are a proven time–tested tool [BR94,BD96,GZ00,SSB01] for that.
- The thrust of our program is about relating different levels of abstraction, and this is the core business of the ASM methodology.

For ASM methodology, our results then just open another application area, in a way somewhat different from that of [BR97,BR98] — we intend not only to use ASMs in order to prove protocols correct, but also to generate their implementations in a uniform way. Formal ASM models for patterns, CrAM and the compilation, in form of executable AsmL code, can be found at [Web].

1 Vocabulary and Assumptions

Any instance of the CrAM is defined wrt the following basic universes: universe $M$ of encoded messages, and universe $O$ of abstract cryptographic objects.

The distinction between $M$ and $O$ is real in concrete cryptographic APIs: every API the authors know about needs to represent a key internally before accessing it for encryption, it is accessible for encryption only in form of an abstract object, say a handle in old–fashioned APIs, an $O$, distinct from the byte–array representation of the key, an $M$ used for communicating it externally. Even
plain byte–objects, such as nonces and hashes, are typically stored internally in an internal form, as an \( O \), distinct in general from the format used for external communication, that of \( M \), say PKCS #7 [PKC93] or PGP messages.

We assume the universe \( O \) to be split into a finite set of pairwise disjoint types \( T_1, \ldots, T_n \). For each type \( T \) we will use, both in patterns and in the CrAM, typed variables \( v^T \) to range over \( T \). We shall also use \textit{raw variables}, which will range over raw messages from \( M \), with metavariable \( r \), as well as metavariables \( o, m \) for objects from \( O, M \) respectively, and metavariable \( v \) for arbitrary object variables.

Each type is associated with one or more of \textit{kinds} such as \texttt{Nonce}, \texttt{Hash}, \texttt{EncryptionKey}, etc. The table bellow lists the kinds, together with a metavariable associated with each kind, to denote variables to range over types of that kind.

\[
\begin{array}{llll}
K & h & e & K \\
k & c & sk & k \\
sh & cr & s & sh \\
n & ek & vk & n \\
a & dk & t & a \\
\end{array}
\]

The names of the kinds listed above are at this level only a heuristic indication of our intentions; every incarnation of the CrAM must provide its meanings for them by providing the associated concrete types.

The type \texttt{RSAES-OAEP-DecryptionKey-1024} is an example of such a concrete type of the kind \texttt{PrivateKey}, embodying a concrete cryptographic decryption primitive RSADP (using default hash function and mask generation algorithm: SHA-1), with a concrete encoding method EME-OAEP, according to PKCS #1 standard [PKC02], and specific key material — 1024 bit long RSA private key. Universe \( M \) of a CrAM instance with this type must also include encodings of all messages encrypted with RSAES-OAEP using 1024 bit long private RSA keys, together with encodings of all 1024 bit long private and public RSA keys.

We assume the following relationships between kinds:

1. Each kind of \texttt{PrivateKey} or \texttt{SharedKey} is, nonexclusively, a \texttt{DecryptionKey} type or a \texttt{SigningKey} type; on the other hand each \texttt{DecryptionKey} or \texttt{SigningKey} type is either a \texttt{PrivateKey} type or a \texttt{SharedKey} type;
2. Each \texttt{PublicKey} or \texttt{SharedKey} type is, nonexclusively, an \texttt{EncryptionKey} type or a \texttt{VerifyingKey} type; on the other hand each \texttt{EncryptionKey} type or \texttt{VerifyingKey} type is either a \texttt{PublicKey} or a \texttt{SharedKey} type;
3. the \texttt{CreatableObject} types are exactly all \texttt{Nonce} types, \texttt{PrivateKey} types and \texttt{SharedKey} types.

The \texttt{PrimitiveValue} types are understood to consist of booleans, bytes and other primitive values needed. The type \texttt{RSAES-OAEP-DecryptionKey-1024} from the example above has the following kinds associated with it: \texttt{PrivateKey}, \texttt{DecryptionKey} and \texttt{CreatableObject}.

The vocabulary contains the procedure signatures listed in the following table, associated to each type \( T \) of appropriate kind (with signatures in terse, but understandable programming style):
<table>
<thead>
<tr>
<th>kind of $T$</th>
<th>“shared interface”</th>
<th>“object interface”</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>$T \text{ decode}_T(M)^\dagger$</td>
<td>$M \text{ encode}_T(T)$</td>
</tr>
<tr>
<td>Tuple</td>
<td>$T \text{ createTuple}_T(M)$†</td>
<td>$\overline{M} \text{ analyzeTuple}_T(T)$</td>
</tr>
<tr>
<td>CreatableObject</td>
<td>$T \text{ create}()$</td>
<td></td>
</tr>
<tr>
<td>Hash</td>
<td>$M \text{ hash}_T(M)$</td>
<td></td>
</tr>
<tr>
<td>PrivateKey</td>
<td>$T' \text{ op}_T(T)$</td>
<td></td>
</tr>
<tr>
<td>EncryptionKey</td>
<td>$M \text{ encrypt}_T(T, M)^\star\dagger$</td>
<td></td>
</tr>
<tr>
<td>DecryptionKey</td>
<td>$M \text{ decrypt}_T(T, M)^\dagger$</td>
<td></td>
</tr>
<tr>
<td>SigningKey</td>
<td>$M \text{ sign}_T(T, M)^\dagger$</td>
<td></td>
</tr>
<tr>
<td>VerifyingKey</td>
<td>$\text{Boolean verify}_T(T, M, M)^\dagger$</td>
<td></td>
</tr>
</tbody>
</table>

where the procedures marked with $\star$ can be nondeterministic, and those marked with $\dagger$ might fail, say in case of unexpected format of an argument; $\overline{M}$ is a shorthand for \texttt{Seq of M} of AsmL.

The intended interpretation of $\text{encode}_T/\text{decode}_T$ is encoding/decoding between objects of type $T$ and encoded messages, according to a formatting standard carried by $T$. Decoding might fail, since the message may not be a valid encoding of some $T$. Thus, given a type $T$ and an encoded message $m \in M$, we either have $\text{encode}_T(\text{decode}_T(m)) = m$, or $\text{decode}_T$ fails at $m$.

Having the same encoding is the only criterion of object identity used in the CrAM.

It would be a nice property of encodings that the domains of $\text{decode}_T$ of different types be disjoint; this would thwart type-flaw attacks on protocols, and it is usually achieved by type-tagging the encodings. For the purpose of this paper, we do not make any such assumptions.

Types of kind Tuple represent different methods of encoding/decoding of message tuples, of fixed or bounded or arbitrary arity. Encoding of a tuple with $\text{createTuple}_T$, can fail, say if an input sequence is of wrong length. Given a tuple object $t$ of type $T$, we assume $\text{encode}_T(\text{createTuple}_T(\text{analyzeTuple}_T(t))) = \text{encode}_T(t)$. Notice the use of $\text{encode}_T$ for establishing equality of abstract objects of type $T$.

Types of kind CreatableObject, those of nonces, private and shared keys, also have a $\text{create}()$ method attached, returning a presumably fresh object.

An object of type $T$ of kind Hash carries a concrete hashing algorithm used in $\text{hash}_T$. Different types may carry different algorithms, say types SHA-1 and MD5 carry SHA-1 [FIP95] and MD5 [Riv92] cryptographic hash algorithms respectively.

The same holds for objects of key-types, but they, in addition to carrying an algorithm, also carry a concrete key to be used in $\text{encrypt}_T, \text{decrypt}_T, \text{sign}_T$ and $\text{verify}_T$. For example, an object of type RSAES-OAEP-Decrypt-1024 carries a 1024 bit long RSA private key, while $\text{decrypt}_{\text{RSAES-OAEP-Decrypt-1024}}$ is exactly the procedure RSAES-OAEP-Decrypt from [PKC02].

For each type $T$ of kind PrivateKey we also have a function $\text{op}_T$, returning an object of attached type $T'$, which has to be of kind PublicKey. Extending the previous example, the type RSAES-OAEP-EncryptionKey-1024 of kinds PublicKey and EncryptionKey, is the codomain of $\text{op}_{\text{RSAES-OAEP-DecryptionKey-1024}}$. 
We also assume that decodings of the codomain of $\text{encrypt}_T$ belong to an $\text{Encryption}$ type, and that decodings of the codomain of a $\text{sign}_T$ belong to a $\text{Signature}$ type.

This is all that we have to assume of key and hash types for the purposes of this paper. It is our understanding that also

- either $\text{encrypt}_T'(\text{opt}_T(K), m))$ fails,
- or $\text{decrypt}_T(K, \text{encrypt}_T'(\text{opt}_T(K), m)) = m,$

and likewise for $\text{verify}_T/\text{sign}_T$ and for shared keys, but we do not use such assumptions in this paper.

It is also common to assume that created objects are “fresh”, encryptions are “hard to break”, signatures are “hard to forge”, and hashes are “hard to invert”. But we cannot even say on this level of abstraction what exactly “fresh”, “hard”, “break”, “forge” or “invert” mean, since CrAM instances at different levels of abstraction, and different instances at the same level of abstraction, will in general interpret these notions differently. In [RRS03] we provide one such interpretation in the context of abstract encryption, where i.e. “hard” means impossible and “invert” means invert. In more concrete instances of the CrAM these notions will obtain different probabilistic polytime interpretations. It is a primary purpose of the CrAM to facilitate relating protocol executions under different interpretations.

**Notation and Mapping to OOP**

In the sequel we use an OOP notation for the above vocabulary, in order to minimize the distance between notation and the AsmL model. In the signatures of the third column of the above table, the “object interface”, the first argument is always of type $T$. It is common in OOP to suppress such an argument, and see it as the “method subject” or “message receiver”. The type–subscript is also superfluous, since the type is implicit in the subject. Thus we shall write $x.\text{encrypt}(m)$ for $\text{encrypt}_T(x, m)$ whenever $x$ is of type $T$ of kind $\text{EncryptionKey}$, and likewise for all signatures in the third column.

Thus our kinds can be seen as $\text{interfaces}$, and the concrete types as concrete $\text{classes}$ implementing some of these interfaces.

The signatures of the second column, the “shared interface”, do not have such designated subjects, they can be seen as declaring “shared” or “static” methods of the attached types. Since a shared method cannot technically be declared in an AsmL interface, we use the following device.

In the AsmL model [Web] the typed variables will also be represented by objects. The type of an object representing $v^T$ knows about $T$, and the shared signatures can become the interface this type has to implement. Thus, if $v^T$ is a variable of type $T$, we shall write $v^T.\text{decode}(m)$ instead of $\text{decode}_T(m)$ whenever we have a variable $v^T$ at hand, what we always will. We shall use all signatures from the second column in this way only. Thus both patterns and the CrAM can remain blissfully ignorant of the concrete types: accessing them only through interfaces, they have no need to mention the concrete types at all.
2 Message Patterns

In this section we revisit the message patterns of [RRS03]. We have related them there to the abstract model of encryption, defined how they can match a message, extracting information from it, and create a message, and shown that appropriate patterns, for any message, can do as well as any AsmL program.

The kinds, interfaces of (types of) cryptographic objects, if used in the patterns, allow us to decouple the patterns from the particular model of abstract encryption, so patterns can apply to any cryptographic model supported by the vocabulary. The patterns can thus be seen as representing the abstract structure of a broad class of cryptographic messages.

This requires slight adaptation of syntax, essentially adopting kinds of cryptographic objects (in addition to extending the syntax with signatures, hashes and agent names). The rules for matching and creation need to be adapted so as to rely only on the functionality of interfaces provided by the kinds, instead of that of the abstract encryption model.

The universality results for patterns wrt abstract encryption can be easily regained by defining the appropriate types.

2.1 Syntax and Semantics

The syntax of message patterns is given in figure 1. It deviates from that of [RRS03] in the following respects:

1. the variables are now typed according to the vocabulary, denoted in the syntax by corresponding metavariables of appropriate kind;
2. the encryption variable of [RRS03] has moved up front, and obtained another role, additional to that of recording an encryption seen/created: its type now determines how to decode an encryption from an external to an internal representation, from an $M$ to an $O$;
3. each compound pattern has been decorated with an appropriate variable for the same purpose;
4. patterns $k : \text{keyOf}(a), h : \text{H}(r), s : S(keyPat, r)$ have been added to deal with agent names, hashes and signatures respectively.

Patterns can now match messages from $M$, and come with a storage split into the following stores:

- *object store* $\sigma$, a finite map of object variables to $O$;
- *message store* $\rho$, a finite map of raw variables to $M$;
- *certificate store* $\nu$, a finite map of encodings of agent names (objects of AgentName types) to encodings of public keys (objects of PublicKey types).

Pattern matching is defined with a relation $\sigma, \rho, \nu, p, m \prec \sigma', \rho'$, where $m$ is a message and $p$ is a pattern. When $\nu$ is understood from context, we write $\sigma, \rho, p, m \prec \sigma', \rho'$. The relation is defined by deduction rules, very similar to those of [RRS03], with the following modifications:
\[
\begin{align*}
\text{varPat} & ::= c \mid K \mid k \mid sh \mid n \mid a \mid e \mid s \mid h \mid r \\
\text{constPat} & ::= c : C \\
\text{opPat} & ::= k : \text{op}(K) \\
\text{keyOfPat} & ::= k : \text{keyOf}(a) \\
\text{encPat} & ::= e : E(\text{keyPat}, p) \\
\text{sigPat} & ::= s : S(\text{keyPat}, r) \\
\text{hashPat} & ::= h : H(r) \\
\text{tuplePat} & ::= t : [p_1, \ldots, p_n] \\
\text{letPat} & ::= \text{let } r = p_1 \text{ in } p_2 \\
\text{keyPat} & ::= k : \text{keyOf}(a) \mid k : \text{op}(K) \mid K \mid k \mid sh
\end{align*}
\]

Fig. 1. Message pattern syntax

1. new rules have been added for the new constructs;
2. \text{encode} and \text{decode} must mediate between internal and external forms of cryptographic objects;
3. equality of objects is determined only by their external representations;
4. rules are adapted to rely only on the functionality provided by types of cryptographic objects and variables used; and
5. the nondeterministic rule for matching tuples has been replaced by its leftmost–deterministic instance for simplicity, since this doesn’t affect the universality results of [RRS03].

The rules are given in figure 2, together with side conditions on their applicability.

Creating messages by patterns is defined by deduction rules as well, proving statements of form

\[
p, \sigma, \rho, \nu \not\vdash m, \sigma'
\]

When $\rho, \nu$ are understood from context, we write $p, \sigma, \rho \not\vdash m, \sigma'$ or $p, \sigma, \not\vdash m, \sigma'$. The rules, given in figure 3, are adapted from the corresponding rules of [RRS03] in very much the same way as the matching rules above.

### 2.2 Recapturing the Relation to Abstract Encryption

In order to recapture the universality properties that the patterns of [RRS03] have wrt the model of abstract encryption explained there, we only have to describe the types used. To keep things simple, we shall not extend the types of abstract messages of [RRS03] here (which is straightforward to do); we shall, for sake of this argument, suppress the new constructs (signatures, hashes and agent names) from patterns instead.
σ, ρ, v, m \downarrow σ, ρ 
σ, ρ, r, m \downarrow σ, ρ 
σ, ρ, v, m \downarrow σ + \{v \mapsto \text{decode}(m)\}, ρ 
σ, ρ, r, m \downarrow σ, ρ + \{r \mapsto m\} 
σ, ρ, c : C, m \downarrow σ + \{c \mapsto C\}, ρ 
σ, ρ, k : \text{op}(K), m \downarrow σ + \{k \mapsto o_k\}, ρ 
σ, ρ, k : \text{keyOf}(a), m \downarrow σ + \{k \mapsto \text{decode}(m)\}, ρ 

σ, ρ, s : S(kp, r), m \downarrow σ' + \{s \mapsto \text{decode}(m)\}, σ' 
σ, ρ, h : H(r), m \downarrow σ + \{h \mapsto \text{decode}(m)\}, ρ 
... \ σ_i-1, \ ρ_i-1, p_i, m_i \downarrow \ σ_i, \ ρ_i ... 
σ, ρ, t : [p_1, ..., p_n], m \downarrow σ_n, ρ_n 

σ, ρ, p_2, m \downarrow σ', ρ', σ'', ρ'' \ σ, ρ, \text{let } r = p_1 \text{ in } p_2, m \downarrow σ'', ρ'' 

kp, σ, ν \downarrow_K o_k, σ' 
σ, ρ, e : \text{E}(kp, p), m \downarrow σ'', ρ'' 

σ(v).\text{encode()} = m 
ρ(r) = m 
v \notin σ 
r \notin ρ 
c \notin σ, C.\text{encode()} = m 
k \notin σ, o_k = σ(K).\text{op}(), 
o_k.\text{encode()} = m 
k \notin σ, m = ν(σ(a).\text{encode}()) 
s \notin σ, o_k.\text{verify}(ρ(r), m) 
h \notin σ, h.\text{hash}(ρ(r)) = m 
t \notin σ, o_t = t.\text{decode}(m) 
\overline{m} = o_t.\text{analyzeTuple}(), 
σ_0 = σ + \{t \mapsto o_t\}, 
ρ_0 = ρ, i = 1, ..., n 
r \notin ρ 

v_k, σ, ν \downarrow_K σ(v_k), σ 
v_k \in σ, v_k = K, k, sh 
k : \text{op}(K), σ, ν \downarrow_K o_k, σ + \{k \mapsto o_k\} 
k \notin σ, K \in σ, o_k = σ(K).\text{op}() 
k : \text{keyOf}(a), σ, ν \downarrow_K o_k, σ + \{k \mapsto o_k\} 
k \notin σ, a \in σ, o_k 
= k.\text{decode}(ν(σ(a).\text{encode}())) 

Fig. 2. Matching patterns to messages

Both universes O and M will coincide with the type Message of [RRS03]. The types of kinds PublicKey, PrivateKey, SharedKey, Nonce, Encryption, Byte or Boolean, Tuple (we do not need any others) will be PublicKey, PrivateKey, SharedKey, Encryption, Nonce, Byte or Boolean, MessageSeq respectively. PrivateKey and SharedKey types will be of DecryptionKey kind, while PublicKey and SharedKey types will be of EncryptionKey kind.

Interpretation of encode_T will be identity of T, while decode_T(m) will be either m for m of type T (since M = O now), or failing otherwise. Thus decode_T will act as a type-checker. In case of type MessageSeq, createTuple is the MessageSeq constructor while analyzeTuple extracts the sequence from a MessageSeq object.

The create methods, where they are needed (in cases of types PrivateKey, SharedKey and Nonce) are implemented by appropriate default constructors.

Under these provisos, as the reader can easily check, the present rules work exactly like the rules of [RRS03], if we disregard the nondeterminism of the tuple-matching rule there, which is not needed for the results of [RRS03].
v, σ ↦ σ(v).encode(), σ
r, σ ↦ ρ(r), σ
c : C, σ ↦ C.encode(), σ + {c ↦ C}
k : op(K), σ ↦ o_k.encode(), σ + {k ↦ o_k}
k : op(K), σ + {K → K.create()} ↦ m, σ'
k : keyOf(a), σ ↦ m_k, σ + {k ↦ k.decode(m_k)}
e : E(kp, p), σ ↦ m_e, σ'' + {e ↦ e.decode(m_e)}
s : S(kp, r), σ ↦ m_s, σ' + {s ↦ s.decode(m_s)}
h : H(r), σ ↦ m_h, σ + {h ↦ h.decode(m_h)}
0, σ, σ'' ↦ m''

Fig. 3. Creating messages through patterns

3 The Cryptographic Abstract Machine

The Cryptographic Abstract Machine is a simple machine with a program, storage, accumulator, input and output. Its storage is decomposed into stores σ, ρ and ν.

More formally, a Cryptographic Abstract Machine is a tuple

$$\mathcal{M} = (\text{prog}, \text{pc}, \sigma, \rho, \nu, \text{in}, \text{out}, \text{acc})$$

where the stores σ, ρ, ν are as in Section 2.1, pc is a program counter holding a nonnegative integer, acc holds a message, and in, out hold either messages or the null object null. The program prog is a sequence of instructions defined in figure 4. Most instructions address variables of some concrete type, what provides them with a specific semantics. It is through the choice of concrete types that the domains of cryptographic objects and messages processed by the machine, as well as cryptographic algorithms used, are determined: we can (and do) run the
<table>
<thead>
<tr>
<th>instruction</th>
<th>update set</th>
<th>condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>INPUT</td>
<td>{in := null, acc := in}</td>
<td>in \neq null</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>{out := acc}</td>
<td>out = null</td>
</tr>
<tr>
<td>LOAD ( r )</td>
<td>{acc := ( \rho(r) )}</td>
<td></td>
</tr>
<tr>
<td>STORE ( r )</td>
<td>{( \rho(r) := acc )}</td>
<td></td>
</tr>
<tr>
<td>STORE CONST ( c )</td>
<td>{( \sigma(c) := C )}</td>
<td></td>
</tr>
<tr>
<td>DECODE ( v )</td>
<td>{( \sigma(v) := v.decode(acc) )}</td>
<td></td>
</tr>
<tr>
<td>ENCODE ( v )</td>
<td>{acc := ( \sigma(v).encode() )}</td>
<td></td>
</tr>
<tr>
<td>CREATE ( cr )</td>
<td>{( \sigma(cr) := cr.create() )}</td>
<td></td>
</tr>
<tr>
<td>ANALYZE TUPLE ( t ) ( \overrightarrow{r} )</td>
<td>{( \rho(r_i) := \sigma(t).analyzeTuple()(i) \mid r_i \in \overrightarrow{r} )}</td>
<td></td>
</tr>
<tr>
<td>CREATE TUPLE ( t ) ( \overrightarrow{r} )</td>
<td>{( \sigma(t) := t.createTuple([\rho(r_i) \mid r_i \in \overrightarrow{r}] )}</td>
<td></td>
</tr>
<tr>
<td>OPPOSITE ( K ) ( k )</td>
<td>{( \sigma(k) := \sigma(K).op() )}</td>
<td></td>
</tr>
<tr>
<td>KEY OF ( K ) ( k )</td>
<td>{acc := ( \nu(acc) )}</td>
<td>acc = ( \rho(r) )</td>
</tr>
<tr>
<td>MATCH ( r )</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>ENCRYPT ( ek )</td>
<td>{acc := ( \sigma(ek).encrypt(acc) )}</td>
<td>( \sigma(\nu k) ) . \text{verify}(( \rho(r) ), acc) )</td>
</tr>
<tr>
<td>DECRYPT ( dk )</td>
<td>{acc := ( \sigma(dk).decrypt(acc) )}</td>
<td></td>
</tr>
<tr>
<td>SIGN ( sk )</td>
<td>{acc := ( \sigma(sk).sign(acc) )}</td>
<td></td>
</tr>
<tr>
<td>VERIFY ( vk ) ( r )</td>
<td>( \emptyset )</td>
<td></td>
</tr>
<tr>
<td>HASH ( h )</td>
<td>{acc := ( h.hash(acc) )}</td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 4.** CrAM instructions

same CrAM programs both with the entirely abstract model of encryption of [RRS03], and with concrete cryptographic APIs, see [Web].

Executing an instruction, if possible, transforms

\[ \mathcal{M} = (\text{prog}, \text{pc}, \sigma, \rho, \nu, \text{in}, \text{out}, \text{acc}) \]

to some

\[ \mathcal{M}' = (\text{prog}, \text{pc}+1, \sigma', \rho', \nu', \text{in}', \text{out}', \text{acc}') \]

by executing the updates associated to each instruction below. There are no branches or jumps in the CrAM, since it is concerned just with analysis and synthesis of messages. If needed, they can be easily added to the machine, or the machine as defined here can be used as a submachine of a machine with some control flow mechanisms. Note that there is no \( \nu' \): in our present model the certificate store is constant.

Executing an instruction can be impossible for one of the following reasons:

1. An explicit side condition of the instruction does not hold.
2. An expression occurring in the updates or in the side conditions cannot be executed because a method is not defined at an argument.
In any of these cases we imagine the machine has failed, aborted (in our implementation we raise an exception).

The fact that one step of execution successfully transforms \( \mathcal{M} \) to \( \mathcal{M}' \) as above will be denoted by \( \mathcal{M} \rightarrow \mathcal{M}' \); the reflexive transitive closure of \( \rightarrow \) will be denoted by \( \overset{*}{\rightarrow} \).

Since some of the operations used in the semantics of CrAM instructions can be nondeterministic, \( \rightarrow \) and \( \overset{*}{\rightarrow} \) are not functions, they are in general just relations.

In the initial state of a CrAM we have \( \text{pc} = 0 \); it is terminated if \( \text{pc} = |\text{prog}| \), i.e. it has executed its entire program without failure. A CrAM in an initial state terminates if there is a terminated CrAM \( \mathcal{M}' \) such that \( \mathcal{M} \overset{*}{\rightarrow} \mathcal{M}' \); we shall say that it terminates with \( m', \sigma', \rho' \) if the value stored in the accumulator is \( m' \), and the object store and the message store of \( \mathcal{M}' \) are respectively \( \sigma', \rho' \). When \( m' \) is not of interest, it will be skipped.

The syntax and the semantics of individual instructions is displayed in figure 4, where the first column provides the syntax, the second column the semantics of instruction execution, given by a set of ASM updates, in terms of \( \sigma, \rho, \nu, \text{in}, \text{out}, \text{acc} \), and the third column the explicit side condition, if any, under which the instruction can execute.

4 Compiling Message Patterns to CrAM

In this section we show how to compile message patterns to CrAM programs, both for matching and for creation. In view of the representation of protocol roles of [RRS03] as sequences of match–create pairs of patterns, we obtain compilation of protocol roles to CrAM code automatically, by pasting and gluing with instructions for input–output.

The intuition, expressed in [RRS03] by saying

A role can be seen as an interactive machine with input and output of messages, with a program given by a sequence of actions. A configuration consists of state of memory (assignment) and program counter (action index).

gets a concrete form in the CrAM, which is now also decoupled from any specific model of encryption, and works with any reasonable model supported by the vocabulary and assumptions.

We show that both compilation algorithms are both correct and complete, which means that CrAM programs are at least as strong as patterns, in any specific incarnation of both.

Compilation for Matching. The compilation algorithm operates in the context of (domains of) current stores \( \sigma, \rho \), and can be seen as a mapping of form

\[
\text{compile} \downharpoonright (p, \sigma, \rho) \mapsto (\text{prog}, \sigma', \rho')
\]
pattern instructions
\[ \text{v} (v \in \sigma) \quad [\text{STORE } r^*, \text{ENCODE } v, \text{MATCH } r^*] \]
\[ \text{v} (v \notin \sigma) \quad [\text{DECODE } v] \]
\[ \text{r} (r \in \rho) \quad [\text{MATCH } r] \]
\[ \text{r} (r \notin \rho) \quad [\text{STORE } r] \]
\[ c : C \quad [\text{STORE } r^*, \text{STORE CONST } c, C, \text{ENCODE } c, \text{MATCH } r^*] \]
\[ k : \text{op}(K) \quad [\text{STORE } r_s, \text{OPPOSITE } K k, \text{ENCODE } k, \text{MATCH } r_s] \]
\[ k : \text{keyOf}(a) \quad [\text{DECODE } k, \text{STORE } r^*, \text{ENCODE } a, \text{KEY OF}, \text{MATCH } r^*] \]
\[ e : E(dk, p) \quad [\text{DECODE } e, \text{DECRYPT } dk] + \text{compile}(\{p, \sigma + \{e\}, \rho\}) \]
\[ s : S(vk, r) \quad [\text{DECODE } s, \text{VERIFY } vk r] \]
\[ s : S(k : \text{op}(K), r) \quad [\text{DECODE } s, \text{OPPOSITE } K k, \text{VERIFY } k r] \]
\[ s : S(k : \text{keyOf}(a), r) \quad [\text{DECODE } s, \text{STORE } r^*, \text{ENCODE } a, \text{KEY OF}, \text{DECODE } k, \text{LOAD } r^*, \text{VERIFY } k r] \]
\[ h : H(r) \quad [\text{DECODE } h, \text{STORE } r^*, \text{LOAD } r, \text{HASH } h, \text{MATCH } r^*] \]
\[ t : [p_1, \ldots, p_n] \quad [\text{DECODE } t, \text{ANALYZE} \text{ TUPLE } t r^3] + \sum_{i=1}^n (\text{LOAD } r_i^*) + \text{compile}(\{p_i, \sigma_i, \rho_i\}) \]
\[
\text{let } r = p_1 \text{ in } p_2 \quad \text{compile}(\{p_2, \sigma, \rho\} + \text{LOAD } r) + \text{compile}(\{p_1, \sigma + \Delta\sigma_2, \rho + \Delta\rho_2\})
\]

Fig. 5. Compilation of a matching pattern to CrAM instructions

If executing the compiled program would overwrite some variable already in \(\sigma\) or \(\rho\), compilation is assumed to fail. Eg. if one of the variables designated in the patterns, ones shown to the left of ‘:’, is in \(\sigma\), then compilation fails.

The inductive clauses defining the compilation algorithm are given in the figure 5. Domains \(\sigma', \rho'\) are domains of a storage resulting from successfully executing a program given by the table, using initial storage with domains \(\sigma, \rho\).

Often the result of compilation involves some scratch raw variables, denoted in the code by \(r^s\). By convention, every occurrence of a scratch variable is fresh, occurring nowhere in the pattern, and nowhere else in the compiled code except for the places indicated. This ensures that scratch variables invoked for a part of a pattern cannot interfere with compilation or interpretation of other parts.

It is easy to check that neither the rules nor the compiled code for matching invoke any possibly nondeterministic operation; the result of matching, if any, is uniquely determined.

Correctness and completeness of the compilation is established by

**Theorem 1.** Let \(p\) be a pattern, \(\sigma, \rho, \nu\) respective assignments, \(m\) a message. Then we have \(\sigma, \rho, \nu, p, m \not\vdash \sigma', \rho'\) if and only if the corresponding CrAM

\[
M = (\text{compile}((p, \sigma, \rho, \nu)), 0, \sigma, \rho, \nu, \text{null, null, null, } m)
\]

terminates in a state with \(\sigma', \rho''\), where \(\rho''\) is an extension of \(\rho'\) with some scratch variables not occurring in \(p\).
pattern

\[ v \in \sigma \] \quad \text{\{ENCODE } v \} \]

\[ cr \in \sigma \quad \text{\{CREATE } cr, \text{ENCODE } cr \} \]

\[ r \in \sigma \quad \text{\{LOAD } r \} \]

\[ c : C \quad \text{\{STORE } \text{CONST } c, C, \text{ENCODE } c \} \]

\[ k : \text{op}(K) \quad \text{compile}^\uparrow_K (k : \text{op}(K), \sigma) + \text{[ENCODE } k] \]

\[ k : \text{keyOf}(a) \quad \text{compile}^\uparrow_K (k : \text{keyOf}(a), \sigma) \]

\[ e : E(kp, p) \quad \text{compile}^\uparrow_K (kp, \sigma) + \text{compile}^\uparrow((p, \sigma + \sigma_{kp})) + \]

\[ \text{[ ENCRYPT keyVar}(kp), \text{DECODE } e] \]

\[ s : S(kp, r) \quad \text{compile}^\uparrow_K (kp, \sigma) + \text{[LOAD } r, \text{SIGN keyVar}(kp), \text{DECODE } s] \]

\[ h : H(r) \quad \text{[LOAD } r, \text{HASH } h, \text{DECODE } h] \]

\[ t : [p_1, \ldots, p_n] \quad \sum_{i=1}^{n} (\text{compile}^\uparrow((p_i, \sigma_i)) + \text{[STORE } r_i^*]) \]

\[ + \text{[CREATE TUPLE } t \vec{r} + \text{ENCODE } t] \]

let \( r = p_1 \) in \( p_2 \)

\[ \text{compile}^\uparrow((p_1, \sigma) + \text{[STORE } r] + \text{compile}^\uparrow((p_2, \sigma + \sigma_1)) \]

keyVar\((kp)\) = \{

\[
 kp = k, K, sh \\
 k \\
 kp = k : \text{op}(K), k : \text{keyOf}(a)
\]

\[ k_v = \text{[ENCODE } \} \quad (k_v \in \sigma), \quad k_v = K, k, sh \]

\[ cr_k = \text{CREATE } cr \]

\[ k : \text{op}(K) \quad \text{compile}^\uparrow_K (K, \sigma) + \text{[OPPOSITE } K k] \]

\[ k : \text{keyOf}(a) \quad \text{[ENCODE } a, \text{KEY OF, DECODE } k] \]

\[ \text{Fig. 6. Compilation of creation pattern into CrAM instructions} \]

The proof proceeds by induction over definition of \( \text{compile}^\uparrow() \), and consists in tedious, though straightforward examination of cases. The reader can try a few cases herself, or find a reasonably expanded proof in the full version of this paper [Web]. We intend to present a fully machine-verified proof elsewhere.

**Compilation for Creation.** The compilation algorithm can be seen as a mapping of form

\[ \text{compile}^\uparrow((p, \sigma) \mapsto (prog, \sigma')) \]

under the same conventions as the compilation algorithm for matching. It is presented in figure 6, which should be read in the same way as figure 5. Again,
if executing the compiled program would overwrite some variable already in $\sigma$, compilation is assumed to fail.

The execution of the compiled program can be nondeterministic, since it could use potentially nondeterministic procedures create, encrypt and sign.

The correctness and completeness of the algorithm is established by

Theorem 2. Let p be a pattern, $\sigma, \rho, \nu$ appropriate assignments, and m a message. Then we have $p, \sigma, \rho, \nu \not\rightarrow m', \sigma'$ if and only if the corresponding CrAM

$$M = (\text{compile}(p, \sigma, \rho, \nu), 0, \sigma, \rho, \nu, \text{null}, \text{null}, m)$$

can terminate in a state with $\text{acc} = m'$, $\sigma'$ and $\rho'$, with $\rho'$ extension of $\rho$ with some scratch variables not occurring in $p$.

The proof also proceeds by induction over definition of $\text{compile}(\cdot)$, and a reasonably expanded proof can be found in the full version of the paper, see [Web].

References


Modeling Discretely Timed Systems
Using Different Magnitudes of Non-standard Reals

Heinrich Rust
Lehrstuhl Software-Systemtechnik, BTU Cottbus
Postfach 101344, D-03013 Cottbus, Germany
rust@informatik.tu-cottbus.de
Tel.: +49(355)69-3801, Fax.: -3810

Abstract. Non-standard reals come in a number of magnitudes which is dense
and which is bounded neither from the top nor from the bottom. This collection
of magnitudes can be put to good use when a system is to be modeled at different
time scales. As an application, we present quantitatively timed models of digital
circuits, we discuss transient states of the system, and we show that under some
plausible conditions, no tedious cycle-counting is necessary in order to establish
freedom of hazards.

1 Introduction

Consider the following scenario: A flip-flop is wired up so that with every rising edge of
the clock, it changes its state. Modeling time using the real numbers and voltage levels
using the numbers zero and one, we assume that the clock line changes its value from 0
to 1 at every even number, and from 1 to 0 at every odd number, and the specification
for the flip flop is that there is an infinitesimal $\epsilon$ so that the output of the flip flop changes
from $x$ to $1 - x$ from time $n$ to $n + \epsilon$ for $n \in \mathbb{N}$ and stays constant from $n + \epsilon$ to $n + 1$,
where the $\epsilon$ allows for the reaction time of the flip flop. The implementation for this
specification would be a set of interconnected digital gates with their associated timings.
The timings would naturally be expressed by infinitesimals, which expresses that we
assume that the gates react very quickly in comparison to the clock changes of the flip
flop. In this context, it would be convenient to assign different infinitesimals as timings
to different gates, or even to be unspecific about the exact timing values, which would
express that we do not know the relative speeds of the circuits, but only that they are all
far quicker than the global clock.

While at first sight, a constant infinitesimal step width for the time might seem to
make such models difficult, this is not the case. It is natural to assume that the timings of
all the gates, while infinitesimal, belong to the same magnitude, and that their differences
are either exactly zero or again of the same magnitude, because we can use just standard
real numbers for expressing the timings, scaled by some freely chosen real number, where
the latter is typically an infinitesimal. The only inconvenience using this approach is that
the possible non-determinism with respect to the timings of different gates must be made
explicit and can not stay implicit as with other strategies in which non-determinism is
the default.

© Springer-Verlag Berlin Heidelberg 2004
Considering the system behaviour on a time scale where the gate delays are irrelevant, all infinitesimal delays would be considered to be negligible. Why should one distinguish them from zero? Because a digital system can be considered on different time scales. If one wishes to consider the system sometimes also on a time scale where gate delays are relevant, modeling them with some non-zero number is necessary. By using numbers of a smaller magnitude, one can use one model of time to express timing aspects of the system on widely differing time scales.

There are other approaches to timed systems with different time scales. One might, for example, use Esterel, VHDL, LUSTRE, StateCharts or some other formalism based on the “synchronous approach” to modeling timed systems [4,14,11,13,12]. Here, two time scales can be distinguished: The larger one can be used explicitly during the modeling, by defining how much time can be spent between discrete macro-steps; the smaller one is used in the definition of the semantics of the formalism in order to define, via a sequence of micro-steps which use negligible time, what happens during a discrete macro-step. If system timings are to be modeled on several time scales explicitly, this is difficult to do with such formalisms, the number of scales is fixed to two, and it means that different modeling paradigms have to be used on the two scales. The current approach avoids these problems.

Another possibility would be to use the standard reals or even the integers in order to model all the timing constants on all time scales, as might be done with all of the the synchronous formalisms mentioned above, or with timed automata and similar formalisms [1,5]. The problem here is that it cannot be expressed by the choice of the numbers that they belong to different “magnitudes”. The non-standard numbers allow to express this directly.

Using non-standard means, the concept “$x$ and $y$ belong to the same magnitude” can be formalized as “$\frac{x}{y}$ is defined and is neither infinitesimal nor unlimited”, or (equivalently) as “there are standard $m, n \in \mathbb{N}$ such that $m \cdot |x| > |y|$ and $n \cdot |y| > |x|$”.

In the current context, we take as infinitesimal step width an infinitesimal which is smaller in magnitude than the timings of the gates; the error introduced by this discretization is smaller in magnitude than the timings.

This idea is described in some detail in this paper.

2 Preliminaries

Non-standard time Abstract State Machines (NTASMS) [17,18] are an approach to use the discrete semantics of Gurevich’s Abstract State Machines [6,7,8] mainly unchanged in order to model quantitatively timed systems. They are proposed as a simpler alternative to the use of a dense model of time. The latter has been used in the context of ASMs by different authors [10,9,3,2].

The main idea we propose is to assume that each step of the discrete ASM uses the same time width, $dt$: The value of a symbol $\texttt{now}$, interpreted as the current time, is incremented in each step by $dt$. In order to be able to model general real-time behavior with sufficient accuracy, this $dt$ should better be small. We consider the use of a positive infinitesimal number for this. Infinitesimals, while commonly used when the calculus was invented in the 17th century, fell into disrepute because of difficulties of finding consistent
formalizations; in the last century, Robinson [16] found, using model-theoretic means, a sufficient formalization, a variant of which was put into axiomatic form by Nelson [15]. Nelson’s idea is to distinguish, in the context of Zermelo-Fraenkel set theory, standard and non-standard sets, where three new axioms describe the properties of a new predicate ‘standard’. In Nelson’s axiomatization the well known real numbers have simply a bit more structure than what is used in classical mathematics: All the entities known from classical mathematics are standard, but Nelson’s axioms shows that there are also non-standard numbers, for example: there exists a strictly positive real number which is strictly smaller than all strictly positive standard real numbers. Note that the new predicate is, in general, not set-forming. For example, there is no set which contains exactly the standard reals.

Numbers which are smaller in absolute value than all strictly positive standard numbers are called infinitesimals, and they correspond well to a (conceptually cleaned up) version of the intuitions which Leibniz and the early users of the calculus seem to have followed. The only standard infinitesimal is zero. The inverse of a non-zero infinitesimal is unlimited; this concept is related to the intuitive understanding of infinity, but not to its set-theoretical formalization: Each natural number is finite, but there exist unlimited natural numbers. A real number is appreciable if it is not infinitesimal and its absolute value is smaller than some standard real number.

### 3 The Magnitude Concept

**Definition 1.** Two real numbers \( x \) and \( y \) belong to the same magnitude, which we abbreviate as \( x \sim y \), if \( \frac{x}{y} \) is defined and appreciable.

The definition implies that neither \( x \) nor \( y \) may be zero. Understood as a relation on non-zero numbers, magnitude is obviously symmetric, transitive and reflexive, which means that is an equivalence relation. For simplicity, we also call the induced equivalence classes magnitudes. Note that zero does not belong to a magnitude by this definition (which is a technical decision), and note that they are not sets – the use of the ‘standard’ predicate in their definition is essential. For example, the magnitude to which the number 1 belongs are just the appreciable numbers.

We make some simple consequences from the definition explicit:

Scaling a number by a non-zero standard number does not change its magnitude, and after both of a pair of numbers of the same magnitudes are scaled by the same non-zero number, the results are again of the same magnitude:

**Proposition 1.** For \( y \in \mathbb{R} \) and standard non-zero \( x \in \mathbb{R} \), we have \( y \sim x \cdot y \).

For \( x, y, z \in \mathbb{R} \) with \( y \sim z \), \( x \neq 0 \), we have \( x \cdot y \sim x \cdot z \).

The following proposition is important for the case that a system is designed using a standard number of standard timing constants and is, as a whole, scaled to some other magnitude. In some cases, derived timing behavior can be described by values computed by via linear combination of design timing with standard factors; the following proposition is applicable.
Proposition 2. Let $C'$ denote a class containing only standard non-zero real numbers, and let $C$ be defined by $x \in C \leftrightarrow \exists (x' \in C') : x = x' \ast a$ for some fixed non-zero $a \in \mathbb{R}$, i.e. all elements of $C$ are scaled by the same (not necessarily standard) $a$. Then for any $y$ which can be constructed as a linear combination from elements of $C$ with standard factors and with a standard number of summands, we have $y = 0 \lor \forall (z \in C') : y \sim z$.

The proposition tells us that under some plausible assumptions on design constants of a system, a linear combination of such values will either be equal to zero or belong to the same magnitude as the timing constants. Note that if the number of summands would be allowed to be non-standard, the result of a linear combination might be of a larger magnitude than the summands: Consider an unlimited number of 1’s added together: the sum is unlimited, which belongs to another magnitude than 1.

Proof. All standard values but zero belong to the same magnitude. This implies that also the scaled elements in $C$ all belong to the same magnitude. Considering the multiplications in the linear combination, note that multiplying a number by a standard number does not change its magnitude, or it yields zero. In our case, the product can also be expressed as the scaling factor $a$ times a standard number. The sum of a standard number of such elements can be expressed, by factoring out $a$, as the product of $a$ and a standard number. If the latter number is not zero, the product is of the same magnitude as the originally scaled numbers, since it is a standard number scaled by the same $a$ as used for scaling the original elements.

We define an order $<$ between magnitudes (understood as equivalence classes) via the normal order $<$ of absolute values of their representatives.

Proposition 3. With respect to the magnitude order $<$, the collection of magnitudes is dense and unbounded.

Proof. Denseness: Take two positive Representatives of two different magnitudes, and call the smaller one $x$ and the larger one $y$. Consider $z = \sqrt{x} \ast y$. It is a standard result that $z$ is located between $x$ and $y$. We will prove that both $\frac{z}{x}$ and $\frac{y}{z}$ are unlimited, which implies that the magnitude to which $z$ belongs lies between those of $x$ and $y$.

Since $x$ and $y$ belong to different magnitudes, $\frac{y}{x}$ is unlimited; this implies (use contradiction) that also $\sqrt{\frac{y}{x}}$ is unlimited.

Since $\frac{z}{x} = \frac{\sqrt{x} \ast y}{x} = \frac{y}{\sqrt{x}}$, where the latter is unlimited, $z$ belongs to a strictly larger magnitude than $x$. And since $\frac{y}{z} = \frac{y}{\sqrt{x} \ast y} = \frac{\sqrt{y}}{\sqrt{x}}$, $y$ belongs to a strictly larger magnitude than $z$.

Unboundedness: For any unlimited number $u$ and for nonzero $x$, $u \ast x$ obviously belongs to a strictly larger magnitude than $x$, and $\frac{z}{u}$ obviously belongs to a strictly smaller magnitude than $x$.

Proposition 3 has convenient consequences for our ability to choose different time scales for describing phenomena of timed systems, since given any system in the description of which a finite number of time scales is used, if a time scale is needed which is larger or smaller than all already employed, or which lies between any two already used, we are assured that such a time scale exists.
falling_edge(C) = def
  LOCAL old
  IF old ≠ C THEN old := C
  VALUE (C < old)
Counter = def
  IF falling_edge(clock) THEN value := (value + 1) mod 8

Fig. 1. Rule describing a 4-bit-counter

The following proposition encapsulates the insight that the truth of a strict relation between two numbers does not depend on errors which are smaller than the magnitude of the difference of the numbers:

**Proposition 4.** Consider two numbers $x$ and $y$ with $x > y$. Then, adding any number smaller in magnitude than $x - y$ to either $x$ or $y$ does not change the relation.

This is a trivial corollary of the fact that in order to change the truth value of $x > y$ by addition of a number $z$ to one summand, the absolute value of $z$ must be at least $x - y$.

### 4 Rule Schemes and the Ripple Counter Example

For the description of systems with a repetitive structure, it is convenient to use rule schemes. They allow multiple invocations of similar rules working just with different locations. Rule schemes are rules with formal parameters for values and locations. Such a rule scheme sometimes can make good use of non-parameter locations which should be local to itself. We use the keyword ‘LOCAL’ to introduce function symbols which are understood to be interpreted as different functions for different instantiations of the rule scheme. Additionally, we will use rules with VALUE components which can be used in syntactical positions of terms in other rules: The rule bodys are executed in parallel to the rest of the system, and the value of the expression after the VALUE key word is assumed in the position of the rule invocation.

As an example, consider a model of a 3-bit counter which performs one step when the clock has a falling edge. On a high abstraction level, we can express such a counter with the rule in Figure 1: On each falling clock edge, the counter is incremented.

We construct such a counter as a three bit ripple counter consisting of three JK master slave flip flops. Let us first describe the function of such a circuit. If the data inputs $J$ and $K$ of such a flip flop are both set to 1 during a rising edge of the clock, the circuit changes its output $Q$ on the following falling edge of the clock input $T$. If the data inputs are both zero, the flip flop does not change its state; and if the data inputs differ, the output assumes the values of input $J$ on the falling edge of the clock. A high level model of such a circuit is given in Figure 2. The expressions with the symbols rising_edge and falling_edge are assumed to yield ‘true’ if the respective signal change has been detected for the argument symbol. Thus, the symbols can not just represent functions (since not only the current value of $T$ is relevant for the value of the expression, and the current value would be the only value available to a proper ASM function).
The local symbols \( \text{risingT}_J \) and \( \text{risingT}_K \) are used to record the values of the input during rising edges of the clock which have to be known on the next falling edge of the clock. \( T, Q, J \) and \( K \) are formal parameters for the clock, the output and the two data inputs of the circuit.

Using the JK flip flop rule scheme, we can put together a ripple counter scheme and instantiate it in the manner described in Figure 2. The scheme describes how the output of one flip flop is wired up with the clock input of the other, as is characteristic of a ripple counter, and that the data inputs are set to one; the last line is an instantiation of the scheme which uses nullary symbols whose values represent the current clock and the current values of the three flip flops.

The ripple counter differs from the high-level counter mainly in one respect: On a falling clock edge, the high-level counter increments its value register in one step, while the three single-bit registers of the ripple counter which together represent the current value might assume transient intermediate values while a carry is being propagated from the lower to the higher bit positions. For example, the transition from 111 to 000 (for \( q_0, q_1, q_3 \)) goes through the intermediate stages 110 and 100.

The ripple counter scheme can be considered an implementation of the high level counter scheme using the following ideas: (a) In order to relate implementation states to specification states, we use an abstraction function which for a state with \( q_0, q_1, q_2 \) being equal to \( a, b, c \) yields a state with value being equal to \( c \times 4 + b \times 2 + a \). (b) In order to be able to ignore ripple phases in the implementation, we allow that a small proportion of states of the implementation does not correspond to their same-time-counterparts in the specification. Modeling ASM runs as functions from the natural numbers to the state space, we define:

**Definition 2.** A rule \( R \) simulates a rule \( R' \) with positive time resolution \( t \in \mathbb{R} \) and an abstraction function \( f \) from \( R \)-states to \( R' \)-states for runs from a state set \( Q' \): if and only if for each \( R' \)-run \( r' \) starting from some \( q \in Q' \) with some time step width \( dt \), there is an \( R \)-run \( r \) with the same time step width so that in each time interval of length \( t \), the proportion of state pairs \( r(i) \) and \( r'(i) \) for which \( r'(i) \neq f(r(i)) \) is infinitesimal.

**Proposition 5.** Let us call the rule of Figure 2 \( R' \) and that of Figure 1 \( R \). We assume that (a) the non-local parts of the state spaces of the rules are almost identical, only
that of $R'$ has the specific symbol value, and $R$ has specific symbols $q_0$, $q_1$, $q_2$. (b) The local symbols are defined by the given rules. (c) Both state spaces contain a symbol clock the value of which switches cyclically from 1 to 0 and back again (assumed to be driven externally), where the distance between falling clock edges is always larger than some strictly positive real number $t$. (d) The time step width $dt$ is of smaller magnitude than $t$. (e) The abstraction function $f$ maps $R$-states to $R'$-states with identical values of all common non-local symbols, and maps states with $q_0$, $q_1$, $q_2$ equal to $a$, $b$, $c$, to states with value having the value $c \times 4 + b \times 2 + a$, and the only 'old' local symbol of $R'$ has the value of the local symbol in $R$ which records the old value of the clock. (f) $Q'$ is the singleton set of $R'$ states in which the values of the 'old clock' symbol and of the 'clock' symbol are both equal to zero.

Under these conditions, $R$ simulates $R'$ at time resolution $t$ with abstraction $f$ for runs starting from $Q'$.

Proof. We compare some run of $R'$ from the element of $Q'$ where the behaviour of the 'clock' input obeys the condition of the proposition, and consider a run of $R$ from a start position where all local variables are zero, and where 'clock' behaves as in the $R'$ run. Both runs use the same $dt$. We have to show that for any time interval of length $t$, the proportion of non-matching states in the two runs is infinitesimal. (1) Since $t$ is of a larger magnitude than $dt$, the number of states in the interval is unlimited. (2a) Only in ripple phases of the implementation, there are states in the $R$ run which do not match their counterparts in the $R'$ run. (2b) Each ripple phase has a standard finite number of states, and (2c) because the starts of ripple phases are distanced by at least time $t$, each $t$ interval contains parts of at most two ripple phases. Thus, the number of non-matching states in each $t$-interval is standard finite. (3) The proportion of a standard finite number to an unlimited number is an infinitesimal.

5 Making Delays Explicit

A problem of the ripple counter implementation of the high level counter is its dependence on the step width $dt$. If another step width is chosen, the timing does not only change because of the different discretization of some timing constants. This results from the fact that the timing of the flip flop is not modeled explicitly – the flip flop model assumes that after a falling edge on the clock input, it takes just time $dt$ until the output is updated.

The timing might be modeled explicitly in different ways. One strategy which is common in hardware languages is to split the behavior of a circuit into a logical and a timing part. E.g., a NAND which changes its output only after some delay after an input has changed is modeled by the serial connection of an instantaneously working NAND and a circuit only representing the delay.

Hardware description languages commonly allow to express several forms of delay mechanisms. Two common ones are inertial delay and transport delay. The specific semantics differ for different formalizations, but the basic idea is that a change at the input of an inertial delay circuit cancels a change event for the output which has been scheduled but not yet performed, while for transport delay, no such cancellation takes place.
InertialDelay(In, Delay, Out) =def
LOCAL old_In, next_change_time
  || old_In := In
  || IF old_In ≠ In THEN next_change_time := now+Delay
     ELIF now ≥ next_change_time THEN next_change_time := ∞
  || IF now ≥ next_change_time THEN Out := old_In
TransportDelay(In, Delay, Out) =def
LOCAL old_In, change
  || old_In := In
  || IF old_In ≠ In THEN change(In, now+Delay) := true
  || CHOOSE s,t:change(s,t) AND t ≤ now IN
     Out := s || change(s,t) := false

Fig. 3. Rule schemes expressing two forms of delay

To be specific, we describe the two forms of delay by two rule schemes with formal parameters for the input signal, the delay time and the output signal, both with two local symbols: one for detecting changes of the input signal, and one for recording information for one (InertialDelay) or several (TransportDelay) scheduled changes. Figure 3 gives rules implementing these forms of delay.

Let us first consider rule scheme InertialDelay. It has three synchronous branches: The first just records the current value of the input so that in the next cycle, an input signal change can be detected. The second updates the internal information about scheduled changes; and the third performs an update if its time has come. Rule scheme TransportDelay is similar, but here, the update of the internal data structure representing scheduled updates is performed in the third branch for those changes which have just been performed. Note that in the third synchronous branch of TransportDelay, if there exists a pair of values \( s \) and \( t \) as in the condition, there is just one such pair, which is then chosen and processed in the THEN branch.

The important point about an inertial delay is that at its output, pulses are never shorter than the delay time:

**Proposition 6.** Consider a location \( l \) which is written at most by an instantiation of rule scheme InertialDelay. Then, between value changes of \( l \), at least the delay time of the instantiation passes.

**Proof.** Consider a system with an instantiation of InertialDelay so that the output location \( l \) of that instantiation is not written by any other rule. Consider any run \( q \) of this system and a position \( i ∈ \mathbb{N}, i > 0 \) with a value change at \( l \), i.e. a position which fulfills \( q(i)(l) ≠ q(i−1)(l) \).

We use an induction to prove an invariant for the value of next_change_time:

**Claimed Invariant.** From position \( i \) on, the value of the symbol next_change_time local to the rule scheme instantiation considered will never be smaller than \( t = q(i−1)(now+Delay) = (i−1) * dt + Delay \).

**Induction Start.** For position \( i \), inspection of the rule scheme shows that the occurrence of an update of location \( l \) at position \( i \) in the run implies that at that position, the
JKFF_D(J,K,T,D,Q) =\text{def} \\
\text{LOCAL internal}_Q \\
JKFF(J,K,T,\text{internal}_Q) \mid \mid \text{InertialDelay(\text{internal}_Q,D,Q)}

Fig. 4. A JK-master-slave flipflop with explicit delay

value of the symbol \text{next\_change\_time} local to the rule scheme instantiation considered must either be $\infty$ (if no input change has just been detected at position $i-1$ in the run) or $t$ (as defined above, if an input change has been detected at position $i-1$). It can not be smaller.

**Induction Step.** For a position $j > i$, either the value of \text{next\_change\_time} did not change in the last step (which implies the claim because of the induction assumption), or it has just been set to $(j-1) \times dt + \text{Delay}$ because of a detection of an input change, which is larger than $t$ because of $j > i$.

The first moment at which the value of the output location might change is $dt$ after the first moment when $\text{now} \geq t$ holds. Let us call the position at which that condition is fulfilled $j-1$, so that $j$ is the position of the first possible change after that at position $i$. We see that the condition $\text{now} \geq t$ at that moment means $(j-1) \times dt \geq (i-1) \times dt + \text{Delay}$; adding $dt$ on both sides, we see that $j \times dt$, which is the moment of the first possible change after that of $i \times dt$, comes at least by $\text{Delay}$ later than $i \times dt$.

As long as the time interval between changes of the inputs is always longer than the delay time, both delay rules are equivalent. They only differ if during the propagation time for one signal change, the input signal changes anew. For InertialDelay, the propagation of the earlier signal change is canceled; for TransportDelay, each signal change of the input is replayed after the delay (plus a discretization error) on the output.

Let us assume that the delay behavior of a JK flip flop with timing constant $D$ (for the delay from the falling edge of the clock input to the update of the output) is well expressed as an inertial delay. Then we can describe a JK flip flop with explicit timing as in Figure 4.

If a ripple counter is implemented with instances of the rule scheme JKFF_D, the appropriateness of the implementation of the high level counter depends on the relative timing of distances of falling edges of the global clock and the delay time of the flip flop. The common assumption that the delay time of the flip flop is negligible with respect to the clock cycle can be expressed formally by the assumption that the delay time is of a smaller magnitude than the clock cycle; and since we can assume that the step width $dt$ of the underlying non-standard time model is of a smaller scale than the delay time, we ensure that discretization errors are very small in comparison to the delay time.

6 Analyzing a Logical Circuit for Hazards

Another timing-related problem of digital circuits are hazards. Hazards are short-term changes of an output of a circuit after changes of the inputs resulting from gate delays. As
a simple example, consider a circuit computing the logical function \( q := (x \land y) \lor (z \land \neg y) \) from the inputs \( x, y \) and \( z \). Such a circuit can be implemented from two AND gates, a NOT gate and an OR gate as given in Figure 5.

Assuming that all gates react with the same inertial delay of \( \tau \) (for simplicity assumed to be a multiple of \( dt \)) on input changes, we can express this circuit by the instantiation of the rule scheme \( \text{HazardCircuit} \) given the end of the set of ASM definitions in Figure 6.

The definition and use of rule scheme \( \text{UpdateIfChanged} \) expresses the modeling decision that the logical gates are not assumed to perform any work if the next value they compute is equal to the current value of the output wire. Otherwise, the system would show unlimited activity in these components, since in each step, an update would be generated. Another approach dealing with this problem would have been to ignore the locations updated by the gates when it is determined if the system shows finite or infinite activity.

Fig. 5. A circuit leading to a hazard in some situations

\[
\text{UpdateIfChanged}(\text{loc, newVal}) = \begin{cases} \text{def} & \text{IF } \text{loc} \neq \text{newVal} \text{ THEN } \text{loc} := \text{newVal} \\
\text{AND}(X,Y,Q) = \text{def} & \text{UpdateIfChanged}(Q, \min(X,Y)) \\
\text{OR}(X,Y,Q) = \text{def} & \text{UpdateIfChanged}(Q, \max(X,Y)) \\
\text{NOT}(X,Q) = \text{def} & \text{UpdateIfChanged}(Q, 1-X) \\
\text{AND}(X,Y,D,Q) = \text{def} & \\
\text{LOCAL} \text{ internal}_Q \\
\text{AND}(X, Y, \text{internal}_Q) \\
\text{OR}(X, Y, \text{internal}_Q) \\
\text{NOT}(X, \text{internal}_Q) \\
\text{HazardCircuit}(X,Y,Z,Q) = \text{def} & \\
\text{LOCAL} \text{ neg}_Y, O1, O2 \\
\text{\quad NOT}(Y, \tau, \text{neg}_Y) \\
\text{\quad AND}(X, Y, \tau, O1) \\
\text{\quad AND}(\text{neg}_Y, Z, \tau, O2) \\
\text{\quad OR}(O1, O2, \tau, Q) \\
\text{HazardCircuit}(x,y,z,q)
\]
We will now analyze the system and detect a possible hazard. Let us consider the following situation: The inputs $x$, $y$, $z$ are all set to 1, and we wait until the system has stabilized (there is no feedback and there are no active components, so it will stabilize in finite time). The output of the first AND gate is 1; the outputs of the NOT gate and of the second AND gate are 0; and the output of the OR gate is 1.

In a run starting from this situation, the system stays stable as long as none of the inputs change, which implies that the output stays at 1. Now consider that input $y$ changes from 1 to 0 and let the first moment with the new value of $y$ be at time $t$. The logic formula for our system shows that the output should not change in this case. But in fact, a negative pulse of width $\tau$, a hazard, will occur at the output; let us first look at this in a rough manner, disregarding the infinitesimal step widths which might lead to infinitesimal errors:

1. At time $t$, the change of $y$ leads to changes of the outputs of the first AND and of the NOT being scheduled for time $t + \tau + dt$.
2. Around time $t + \tau$, the change of $y$ has propagated through the first AND (making its output 0) and through the NOT, but not yet through the second AND or the OR gate; but in this moment, both inputs of the OR gate are zero, and thus, a change of the output of the circuit from 1 to 0 is scheduled; and both inputs of the second AND gate are 1, so a change of its output from 0 to 1 is scheduled.
3. Around time $t + 2 \times \tau$, both scheduled changes become effective: $q$ becomes 0, and the second input of the OR gate becomes 1, leading to another change of the output being scheduled.
4. Around time $t + 3 \times \tau$, the scheduled change of $q$ is performed, and the system becomes stable again.

This analysis is not exact because it disregards the following facts about our model:

- When a change of the input of a gate with explicit timing takes place, first the untimed (‘instantaneous’) version of the gate computes its output, which needs time $dt$.
- When a change of the output of the untimed gate is detected at its input by the delay rule, its output only changes after $\lceil \frac{\tau}{dt} \rceil \times dt$ time units.

Altogether, a change at an input of a timed circuit can only lead to a change at its output (if at all) after time $\tau + dt$. Since we assume that the time delay $\tau$ is of a larger magnitude than $dt$, this difference can be ignored in most cases. In our specific case, this might just lead to the hazard being an infinitesimal of magnitude $dt$ being shifted or longer than expected.

The following section discusses conditions under which such tedious cycle-counting as just performed is not necessary.

### 7 Modeling Missing Knowledge Explicitly

Let us consider another variant of the timing problem for digital circuits. Mandrioli\textsuperscript{1} points out that the assumption of a constant infinitesimal step width for the work of

\textsuperscript{1} Personal communication.
system components yields a high degree of synchronicity which might be inappropriate in some cases. He specifically mentions models of electronic circuits in which gate delays and other signal propagation times can depend on many factors not modeled explicitly.

As an example for such a case, we consider a situation in which under synchronous execution, no hazards can occur, while under more realistic assumptions about our knowledge about relative signal propagation times of system components, the occurrence of hazards can not be excluded.

Consider again the circuit of Figure 5, but consider now a stable situation resulting from the inputs $x=1$, $y=0$ and $z=1$ and a change of $y$ to 1. Under a synchronous interpretation (or under an interpretation in which all gate delays are identical), no hazard occurs since the output of the first AND gate becomes 1 in this case before the output of the second AND turns to 0; thus, there is always at least one input of the OR gate equal to 1, which means that the output of the OR stays 1 all the time.

But now consider the case in which the first AND takes longer to compute its result than the NOT and the second AND combined. In this case, the output of the first AND turns to 1 only after the output of the second AND has turned to 0, which means that there is some time in which both inputs of the OR gate are zero. If this time is longer than the inertial delay modeling the gate delay of the OR gate, a hazard will occur at its output.

This discrepancy can be resolved by making explicit in our model that knowledge is missing about the relative gate delays of the gates. Firstly, this means that we have to use gate models with explicit timing; and secondly, we have to express that the delays might be different for the different gates. For this case, an NTASM model of the circuit might look like in Figure 7. Here, we make explicit that the delay times of the gates might be different; further declarative restrictions on the different $\tau_i$ might be used to express knowledge about relative timings of the gates which we might possibly have. Note that this description might seem unnecessarily wordy in comparison to models using formalisms in which nondeterminism can be implicit; this is just the price we have to pay if we want nondeterminism to be explicit.

Let us analyze in more detail a condition under which hazards can not occur in the system above if only a single input bit changes. This will illustrate an example of the case where we do not have to count each cycle in order to avoid errors.

**Proposition 7.** Consider the instantiated rule scheme

$$\text{IndependentDelaysHazardCircuit}(x, y, z, q, \tau_1, \tau_2, \tau_3, \tau_4),$$

where the $\tau_i$ result from standard real numbers scaled by some arbitrary non-zero real number, and the step width $dt$ is assumed to be of a smaller magnitude than the $\tau_i$. 
If $\tau_4$ is larger than the absolute value of $\tau_1 + \tau_3 - \tau_2$, hazards resulting from single input bit changes of that circuit can not occur.

**Proof.** From the inputs to the output of the circuit, four different paths are possible, one from the $x$, two from the $y$ and one from the $z$. We only look for hazards occurring from changes of a single input bit when the system is in a stable state. If the $x$ or the $z$ input changes, the change will propagate along its only path until it either stops being propagated at some gate or it reaches the output $q$; this depends on the rest of the system, but is not relevant for hazard generation. Thus, only with changes of $y$, from where two paths lead to the output, hazards can happen.

If some of the single-path inputs $x$ and $z$ are zero during the change of $y$, propagation of the $y$-signal change along the paths which are common with the zero-inputs is blocked at the AND gates, i.e. in this case, at most one, and possibly none, of the inputs of the OR gate changes because of the signal change. In these cases, no hazard can occur. Thus, we only have to deal with the case that $x$ and $z$ are both one, since only in this case, two changes can in general occur at inputs of the OR gate.

Note that in both steady states which are left to be considered, one input of the OR gate is zero and the other is one. The accumulated signal propagation times from $y$ to the inputs of the OR gate are $\tau_2$ for the upper path and $\tau_1 + \tau_3$ for the lower path. If the absolute value of the difference between them is smaller than $\tau_4$, there is no problem, because the inertial delay of the OR gate will buffer possible effects of the first signal change away.

This almost closes the argument; it only rests to be shown why it does not invalidate our argument that we disregarded the infinitesimal errors induced by our discretization of time. For this, we apply Proposition 2 to the $\tau_i$, which fulfill the requirements of the proposition. The proposition implies that $\tau_4 - |\tau_1 + \tau_3 - \tau_2|$ is either zero or of the same magnitude as the $\tau_i$. Under the condition that it is positive (which is the condition considered in the proposition), this means that the difference is of a larger magnitude than $dt$. Applying Proposition 4, we see that additive errors of magnitude $dt$ during the computation of the propagation times will not invalidate the argument.

Arguments like the above about the possibility of ignoring errors of sufficiently small magnitude are only valid if (1) we use strict comparisons of quantities, (2) errors are additive, and (3) we know that the absolute value of the difference of the quantities compared is larger in magnitude than the error; or in short, if the conditions of Proposition 4 are fulfilled.

8 Hazards Resulting from the Infinitesimal Discretization

Consider the circuit of Figure 5 again and an NTASM model for it using untimed gates instead of the timed gates used earlier, as in Figure 8. Considering the initial configuration corresponding to that of the case considered earlier and a change of $y$ from 1 to 0, we see that after time $dt$, the output of the first AND changes from 1 to 0 and that of the NOT from 0 to 1. After time $2 \times dt$ the second AND has changed from 0 to 1 and the OR from 1 to 0, resulting in the start of the hazard. After time $3 \times dt$ the output of the OR changes
UntimedHazardCircuit(X, Y, Z, Q) =def 
LOCAL neg_Y, O1, O2
    || NOT(Y, neg_Y) || AND(X, Y, O1)
    || AND(neg_Y, Z, O2) || OR(O1, O2, Q)
UntimedHazardCircuit(x, y, z, q)

Fig. 8. An untimed hazard circuit

Again, finishing the hazard and leading to the new stable state. Thus, similar to the case considered before, we have a hazard at q, but this time it is only of length \( dt \).

Can infinitesimal hazards like this also happen in circuits in which timing is handled explicitly? This would be an inconvenient counter-intuitive consequence of our model of time which would make it difficult to express refinement of a more complex untimed circuit (e.g., one computing the function \( q := (x \text{ AND } y) \text{ OR } (z \text{ AND NOT } y) \) directly) into components.

Note that also semantical approaches for gates not using discretized time but ‘real’ zero-time steps have analogous problems, in which the output waveform resulting from a simulation might depend on the order of schedulings chosen for the gate activities, possibly resulting in zero-time hazards – so, this problem is not specific to our model of time. In hardware simulation languages based on a classical model of time, this problem is sometimes dealt with by not propagating any output signal changes until the system stabilizes. An example is VHDL [14].

We can identify a modeling strategy under which the problem disappears in our framework, and which is based on the same idea as just described for classical approaches, in the following way: We ensure that on every data path from some wire where hazards might occur to any wire of the system where hazards must not occur, an inertial delay must be passed which has a timing constant larger than the maximal length of the hazard. In this case, the first signal change resulting from a possible hazard will be cancelled by the second signal changed before the first has been propagated to the output. With the strategy used above for modeling timing, i.e. by appending inertial delays of a larger magnitude than the step width \( dt \) to the outputs of the untimed circuits, this condition is fulfilled automatically: At the outputs of circuits with explicitly given timing, no hazards resulting from infinitesimal discretization can occur.

9 Further Work

Further work is necessary in order to help modellers use the non-standard numbers in practice: (a) Step-wise simulation of infinitesimals steps is not feasible. Thus, specific simulation environments have to be developed which have to be able to deal with infinitesimal step widths and an unlimited number of do-nothing steps of infinitesimal length. (b) In order to support machine help for proofs, proof support systems would have to be extended by Nelson’s axioms, and a library of helpful lemmata would have to be described; an example of such a lemma is Proposition 4. The addition of Nelson’s axioms might be complicated by the fact that the new predicate standard does not fulfill all the properties of classical predicates; for example, it has to be reflected in the proof
system that standard is in general not set-forming. (c) General modeling rules have to be collected which show how the different magnitudes of the non-standard reals could best be put to use when timed systems are modeled. These rules would have to be formulated in a way so that lemmata like Proposition 2 could be applied easily to the modeled systems, as we did in Section 7.

10 Summary

It is illustrated how different magnitudes of infinitesimals can be used to make explicit that a system might be considered at different time scales, or at different levels of dynamical abstraction.

There are infinitely many magnitudes, and they form a dense collection which has neither a lower nor an upper bound, i.e. between any two magnitudes, there is always another, and for any magnitude, there is always a larger and always a smaller one. Because of this denseness and unboundedness of magnitudes, we are able to very flexibly choose time scales for different levels of abstraction.

Often, the abstraction that the propagation time of circuits is negligible with respect to the global clock cycle is not appropriate. In this case, magnitudes can be use to separate functional and timing related concerns, by first dealing with functional and logical issues and ignoring timing issues (made explicit by assuming that the relevant timing constants are of different magnitudes), and by afterwards taking timing issues into account, by using estimations of the timing constants which fulfill more realistic conditions.

If it is appropriate to assume that the timing of the gates and the time step width $dt$ are of different magnitudes, reasoning about the system can become quite simple because one does not have to count each single step of width $dt$ as long as the number of ignored steps is standard.

I thank the reviewers for their helpful remarks.

References

Author Index

Börger, Egon 38
Bussche, Jan Van den 61

Cavarra, Alessandra 62

Farahbod, Roozbeh 78
Fisher, Michael 95
Fruja, Nicu G. 127

Glässer, Uwe 78
Glesner, Sabine 144
Gurevich, Yuri 1, 161

Hanisch, Hans-Michael 6
Küster-Filipe, Juliana 62

Langmaack, Hans 20
Lisitsa, Alexei 95

Nanchen, Stanislas 169
Nowack, Antje 186

Riccobene, Elvinia 111
Rosenzweig, Dean 202
Runje, Davor 202
Rust, Heinrich 218

Scandurra, Patrizia 111
Stärk, Robert F. 38, 169

Vajihollahi, Mona 78

Yavorskiy, Rostislav 161