

MODEL CHECKING SUPPORT FOR CoreASM:
MODEL CHECKING DISTRIBUTED ABSTRACT STATE
MACHINES USING SPIN

by

George Ma

Bachelor of Science with First Class Honors, University of Alberta, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© George Ma 2007
SIMON FRASER UNIVERSITY
Summer 2007

All rights reserved. This work may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

APPROVAL

Name: George Ma
Degree: Master of Science
Title of thesis: Model Checking Support for CoreASM: Model Checking Distributed Abstract State Machines Using Spin

Examining Committee: Mr. Bradley Bart
Chair

Dr. Uwe Glaesser, Senior Supervisor

Dr. David Mitchell, Supervisor

Dr. Dirk Beyer, SFU Examiner

Date Approved: _____

Abstract

We present an approach to model checking Abstract State Machines, in the context of a larger project called CoreASM, which aims to provide a comprehensive and extensible tool environment for the design, validation, and verification of systems using the *Abstract State Machine* (ASM) formal methodology. *Model checking* is an automated and efficient formal verification technique that allows us to algorithmically prove properties about state transition systems. This thesis describes the design and implementation of model checking support for CoreASM, thereby enabling formal verification of ASMs. We specify extensions to CoreASM required to support model checking, as well as present a novel procedure for transforming CoreASM specifications into Promela models, which can be checked by the Spin model checker. We also present the results of applying our ASM model checking tool to several non-trivial software specifications.

Keywords: abstract state machines, model checking, distributed systems, formal verification tools, CoreASM

To my parents Guo Liang and Yong Mei Ma

Acknowledgements

First of all, I wish to thank my senior supervisor Uwe Glässer for his guidance and support throughout my Masters studies. I appreciate his confidence in giving me the freedom to explore my research as I desired.

A million thanks go to my friend and colleague Roozbeh Farahbod for helping me in every aspect of my work. He provided a constant supply of encouragement, thoughtful criticism, and good humor.

I thank all my colleagues in the Software Technology Lab, past and present, for providing such a friendly and engaging environment in which to work.

I also wish to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for their financial support during a part of my graduate studies.

Most of all, I thank my family for their enduring love and support, which have shaped and inspired my being.

Contents

Approval	ii
Abstract	iii
Dedication	iv
Acknowledgements	v
Contents	vi
List of Tables	ix
List of Figures	x
1 Introduction	1
1.1 Motivation	1
1.2 Objective and Significance	3
1.3 Organization of Thesis	3
2 Related Work	5
3 Model Checking Concepts	8
3.1 Kripke Structures	9
3.2 Temporal Logic	9
3.2.1 Linear Temporal Logic	10
3.2.2 Other Temporal Logics	11
3.3 Automata Based Model Checking	11

3.3.1	Büchi Automata	12
3.3.2	The Model Checking Problem in Terms of Automata	12
3.3.3	Computing $\mathcal{L}(A_M \times A_{\neg\phi})$	13
3.4	Other Model Checking Approaches	13
3.5	Validation Versus Verification	14
4	Abstract State Machines	15
4.1	Basic Abstract State Machines	15
4.1.1	Signatures and States	15
4.1.2	Locations and Updates	16
4.1.3	ASM Rules	17
4.2	Distributed Abstract State Machines	21
4.3	A Sample ASM Specification: Master-slave Agreement	24
5	Extending CoreASM for Model Checking	27
5.1	CoreASM: An Overview	28
5.1.1	CoreASM Plug-in Framework	33
5.1.2	State Representation in CoreASM	36
5.2	Signature Plug-in	42
5.2.1	Functions	43
5.2.2	Universes and Enumerations	45
5.3	Property Plug-in	47
6	From CoreASM to Promela	49
6.1	Elements, Universes and Backgrounds	51
6.2	Functions	52
6.3	Rules	54
6.3.1	Assignment Rule	54
6.3.2	Skip Rule	54
6.3.3	Block Rule	54
6.3.4	Conditional Rule	55
6.3.5	Choose Rule	56
6.3.6	Forall Rule	57
6.3.7	Map Assignment Rule	58

6.3.8	Macro Rules	58
6.3.9	Program Rules	59
6.4	Expressions	60
6.4.1	Forall and Exists Expressions	60
6.5	LTL Correctness Properties	62
6.6	DASM Simulation Model	64
7	Case Studies	67
7.1	Distributed Termination Detection	67
7.2	FLASH Cache Coherence Protocol	69
7.3	i-Protocol	70
7.4	Discussion	71
8	Conclusion and Future Work	73
8.1	Future Work	74
A	CoreASM Specifications from Case Studies	76
A.1	Distributed Termination Detection	76
A.2	FLASH Cache Coherence Protocol	80
A.3	i-Protocol	95
	Bibliography	106

List of Tables

5.1	CoreASM Plug-in Interfaces	33
5.2	LTL Operators Provided by the Property Plug-in	48
6.1	CoreASM to Promela Operator Conversion	61
6.2	Promela LTL Operators	62
7.1	Distributed Termination Detection Model Checking Results	69
7.2	Flash Cache Coherence Protocol Model Checking Results	70
7.3	iProtocol Model Checking Results	71

List of Figures

4.1	Partially Ordered Runs of the Light Control System	23
5.1	Overall Architecture of CoreASM	29
5.2	Layers and Modules of the CoreASM Engine	29
5.3	Control State ASM of Initializing CoreASM Engine	31
5.4	Control State ASM of Loading a CoreASM Specification	31
5.5	Control State ASM of a STEP command	32
5.6	Core Elements in the Kernel	37
6.1	CoreASM2Promela: Overall Verification Process	50

Chapter 1

Introduction

This thesis presents an approach to model checking Abstract State Machines. This work was done in the context of a larger project called CoreASM [22, 24, 23], which aims to provide a comprehensive and extensible tool environment for the design, validation, and verification of systems using the *Abstract State Machine* (ASM) formal methodology. By allowing the execution of ASM specifications, the CoreASM engine facilitates experimental validation of ASM models. Executing an ASM model allows us to see the general behavior of a system with respect to some input or some specific initial state. However, experimental validation does not allow us to formally verify the overall correctness of a system with respect to all of its behaviors. *Model checking* is an automated and efficient formal verification technique that allow us to algorithmically prove properties about state transition systems. This thesis describes the design and implementation of model checking support for CoreASM, thereby enabling formal verification of Abstract State Machines.

1.1 Motivation

As the adoption of information technology continues to grow, with software pervading many aspects of modern life, the importance of engineering software efficiently and correctly is paramount. Software projects are notorious for exceeding budgeted time and resources and for failing to meet user requirements [16, 28, 1]. Designing high quality software is extremely challenging, and software failures can be very costly [55, 50]. By some estimates, testing accounts for half of the effort spent in software development [37]. Software failure in mission and safety critical systems can be extremely costly and in the worst cases have lead to loss

of human life [47]. These problems have motivated the use of *formal methods* in software engineering. Formal methods are mathematically based techniques for the specification, development, and verification of systems. Applying a formal method to the design of a system produces a precise specification of the system's requirements. One can think of formal specifications as being blueprints for software. By catching and fixing design errors and inconsistencies early in the design process, we avoid dealing with them later in the development process, when the cost of making changes is much higher.

The Abstract State Machine method is one such formal method. An ASM specification is a set of pseudo-code-like rules describing the behavior of an abstract machine. ASMs are a type of state transition system, and thus provide an operational system description. The semantics of ASM is mathematically well defined, yet basic and relatively simple to understand. The state of an ASM is described as a set of *universes* and *functions*. The evolution of the system is specified by a set *rules* which produce *updates*. The distinguishing characteristic of the ASM method is that it allows a designer to specify a system at a *natural* level of abstraction. During system design, a system can be modeled at a conceptual level, considering only the aspects of the system that are relevant to capturing the informal requirements. As more design decisions are made, details can be filled in, thereby refining an abstract model into a more concrete model, which can be implemented in a programming language. The effectiveness of ASM method has been proven through its application in industrial settings, as illustrated in works by Börger, Gurevich, Glässer, and others [4, 33, 9, 2].

While the rigor and structure imposed by using formal methods can help improve the quality of software by removing ambiguities and sharpening understanding of system requirements, software specifications are still ultimately descriptions of algorithms, which need to be tested in some way. Traditional software testing involves producing test cases in an effort to cover as much of a program's behavior as possible. As the complexity of a system increases this task becomes more and more difficult. Test cases are often generated manually in an unsystematic fashion. On the other hand, *formal verification* techniques aim to increase software quality by providing a consistent logical framework in which to reason about the correctness of programs. Within such a framework, a property of a system can actually be *proven*. Model checking is a formal verification technique where a system is modeled as a finite state transition system and a correctness property, given as a temporal logic formula, is checked against this model. A model checking program uses an efficient

search technique to determine if a property is satisfied by a model. If a model does not satisfy a property, a counterexample is produced. We believe that applying model checking to ASMs can be a useful tool in assuring the correctness and quality of software specifications.

1.2 Objective and Significance

The objective of this work is to facilitate model checking of abstract state machines. This work is certainly not the first of its kind and the related work will be discussed in Chapter 2 of this thesis. More specifically, the goal of this work is to provide model checking support for CoreASM specifications by translating CoreASM models into Promela models, which can be verified by the Spin model checker. We present a novel approach to performing this transformation that supports distributed abstract state machines. Spin, the recipient of the ACM System Software Award in 2001, is a widely used automata based model checker that has been used extensively in the design of asynchronous distributed systems [35]. Moreover, we aim to provide a tool that is simple to use and well integrated with the other existing CoreASM tools. This thesis also illustrates the extensibility of CoreASM by presenting specifications to CoreASM plug-ins which allow function signatures and correctness properties to be included as part of a specification. Overall, this work significantly progresses the CoreASM project towards its goal of providing an open-source and platform-independent tool environment for the design, validation, and *verification* of abstract state machines.

1.3 Organization of Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 discusses related works done on model checking abstract state machines and characterizes how this thesis fits in that landscape.
- Chapter 3 introduces basic model checking concepts and describes the verification procedure used by the Spin model checker.
- Chapter 4 serves as a general introduction to Abstract State Machines.
- Chapter 5 introduces the CoreASM project and describes the extensions made to CoreASM to facilitate model checking.

- Chapter 6 describes a novel procedure for translating CoreASM specifications into Promela models.
- Chapter 7 presents results on using our model checking tool.
- Chapter 8 concludes this thesis and discusses possible areas of future work.

Chapter 2

Related Work

A substantial amount of work has been done on model checking software specifications. A notable case study was done on the Traffic Alert and Collision Avoidance System II System Requirements Specification [12]. In this study, requirements specified in the Requirements State Machine Language were translated into input for the Symbolic Model Verifier (SMV) [39], a symbolic model checker which uses binary decision diagrams (BDDs) to represent functions, which made it possible to verify several safety and correctness properties of the system. This study illustrates the feasibility of model checking large software specifications of real world systems.

Del Castillo and Winter present an approach for model checking abstract state machines in [18]. In this work, specifications written in ASM-SL, the ASM language for the ASM Workbench tool [17], are translated into input for the SMV model checker. Their translation scheme supports most basic ASM rules (excluding **import** and **choose**) and arbitrary n -ary functions. The translation works by unfolding all rules and functions into basic updates to state variables. This approach is applied to the verification of the Stanford FLASH Cache Coherence Protocol and the Production Cell system. Moreover, Winter's PhD thesis [56] presents an extension to the translation procedure to produce input for a model checker based on multi-way decision graphs (MDGs), which subsume BDDs. This extension supports the use of abstract data types. A case study using the approach is presented in [31].

Gargantini and Riccobenne present a method for model checking ASMs using the Spin model checker [29]. In this work, counterexamples produced by Spin are transformed to generate test cases. Specifications written for the AsmGofer tool [51] are translated into Promela, the input language for the Spin model checker [35]. Spin is an explicit state

model checker that uses an optimized depth first search algorithm to perform verification. Gargantini’s work only supports basic single agent ASMs and a restricted subset of the standard ASM language. The rules supported are *assignment* and *conditional* rules. Also, only *nullary* functions are supported. These are considerable limitations on the power and flexibility of an ASM specification language.

In more recent work, Tang and Ternovska present a method for bounded model checking of ASMs using Answer Set Programming [53]. In this work, ASM-SL specifications are translated into answer set programs. Answer Set Programming is a relatively novel declarative logical programming paradigm, which is based on stable model semantics, for solving combinatorial search problems. The translation supports all basic ASM rules, including finite use of *import* statements, as well as arbitrary n -ary functions.

Martin Kardos has also recently developed a model checker for the Abstract State Machine Language (AsmL) [38]. AsmL [45] was developed by the Foundations of Software Engineering (FSE) group at Microsoft Research and is based on the .NET runtime environment. Kardos has developed a native model checker which works directly with AsmL specifications (instead of translating into input for an existing model checker). His model checker uses the explicit state exploration algorithm described in [5], which is similar to the algorithm used by Spin. The effectiveness of Kardos’ model checker has not yet been shown for non-trivial specifications.

It is also worthwhile to note that the FSE group at Microsoft has also developed a model based testing tool for AsmL called Spec Explorer [46]. However, strictly speaking, Spec Explorer is not a complete model checking tool, as it generates a finite state machine which is only an approximation of an original system model. Spec Explorer’s model exploration technique uses heuristics based sampling. Moreover, it is only capable of verifying specific predefined classes of temporal properties.

The work presented in this thesis is similar to the previously mentioned work done by Gargantini and Riccobenne, as our work also uses Spin to model check ASMs. However, the translation procedure described in this thesis extends their work in several important regards:

- Support for all basic ASM rules, save for **import** .
- Support for arbitrary n -ary functions.
- Support for *distributed* abstract state machines.

Some of the works mentioned this chapter use the ASM-SL specification language. ASM-SL and its supporting tool environment the ASM Workbench are no longer being developed or maintained, nor is the ASM Workbench publicly available. This lack of tool support makes practical use of ASM-SL as a specification language much more difficult, as there is no way to experimentally validate models specified using ASM-SL. On the other hand, the open-source CoreASM tool environment is actively maintained and is being used by various research groups around the world.

Chapter 3

Model Checking Concepts

Model checking is a method for algorithmically verifying systems. In model checking, every possible execution path of a program can be computed (directly or indirectly), allowing a full state-space exploration (search) of a program. Thus, a given property can be checked to see whether it holds true in every possible state of the system. More formally, the model checking problem can be stated as: given a model (Kripke structure) M , an initial state s , and some temporal logical property ϕ , decide if $M, s \models \phi$.

The key challenge to the wide application of model checking software is the state space explosion problem. While model checking has been successfully applied to hardware systems in industrial settings, its adoption in the software world has been much less rapid because the state space of software systems is frequently several orders of magnitude larger than those of hardware systems. One strategy for addressing the state space explosion problem is through the use of abstraction. By modeling a system at a level that is only relevant to the properties being checked, the size of the state space can be greatly reduced. Based on this reasoning, abstract state machines are particularly well suited for model checking software systems. However, one should not conflate this general idea of design abstraction, with the formal notion of abstraction in model checking presented by Clarke et al. in [15].

Generally, model checking approaches can be divided into two categories, logical approaches based on fixed point computation, and automata theoretic approaches based on language containment. Each of these strategies has its advantages and disadvantages. Automata based model checking will be the focus of this chapter, as it is the model checking strategy employed by Spin, the model checker used in this work. Spin was chosen because of the high level constructs offered by its input language Promela, and because of its strong

reputation in the verification of protocols and reactive systems.

The remainder of this chapter first presents fundamental model checking concepts, namely Kripke structures and temporal logic, followed by a detailed explanation of the automata-theoretic approach to model checking. Other model checking techniques will also be discussed briefly.

3.1 Kripke Structures

Kripke structures provide the mathematical framework for reasoning about model checking algorithms. A Kripke structure [41], which describes a state transition system, is a four-tuple $M = (S, I, R, L)$ where:

- S is a countable set of states.
- $I \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation. This relation is total.
- $L : S \rightarrow 2^P$ is a labeling function, where P is a set of atomic propositions. Each state is labeled with the atomic propositions which are true in that state.

A path π in a Kripke structure is a (possibly infinite) sequence of states (s_0, s_1, s_2, \dots) such that for each $i \geq 0$, $(s_i, s_{i+1}) \in R$. The notation $\pi(i)$ denotes the i -th state (s_i) of the path, while π^i denotes the path suffix starting at s_i . A path is *initialized* if $\pi(0) \in I$.

3.2 Temporal Logic

Model checking requires a suitable logic for specifying properties of state transition systems. Basic propositional logic formulas can only describe a single fixed state. In order to reason about computations, we require a dynamic state logic that will allow us to describe the temporal properties over different states of a system. We focus our discussion on the temporal logic used by the Spin model checker, Linear Temporal Logic (LTL) [48]. Other temporal logics will also be discussed briefly.

3.2.1 Linear Temporal Logic

Linear temporal logic models time as a sequence of states which extends infinitely into the future. As the future is not determined, we wish to reason about all possible paths. LTL extends standard propositional logic with temporal operators. The syntax of LTL formulas is defined as follows:

$$\phi ::= \top \mid \perp \mid p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \rightarrow \phi) \mid (\mathbf{X} \phi) \mid (\mathbf{F} \phi) \mid (\mathbf{G} \phi) \mid (\phi \mathbf{U} \phi)$$

where p is some atomic proposition, and \mathbf{X} (*next*), \mathbf{F} (*eventually*), \mathbf{G} (*always*), and \mathbf{U} (*until*) are temporal connectives. The relation that defines whether a path satisfies a LTL formula ($\pi \models \phi$) is defined as follows:

- $\pi \models \top$
- $\pi \not\models \perp$
- $\pi \models p$ iff $p \in L(s_0)$
- $\pi \models \neg\phi$ iff $\pi \not\models \phi$
- $\pi \models \phi_1 \wedge \phi_2$ iff $\pi \models \phi_1$ and $\pi \models \phi_2$
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$
- $\pi \models \phi_1 \rightarrow \phi_2$ iff $\pi \models \phi_2$ whenever $\pi \models \phi_1$
- $\pi \models \mathbf{X} \phi$ iff $\pi^1 \models \phi$
- $\pi \models \mathbf{G} \phi$ iff for all $i \geq 0, \pi^i \models \phi$
- $\pi \models \mathbf{F} \phi$ iff there is some $i \geq 0$ such that $\pi^i \models \phi$
- $\pi \models \phi \mathbf{U} \psi$ iff there is some $i \geq 0$ such that $\pi^i \models \psi$ and for all $j = 0, \dots, i-1$ we have $\pi^j \models \phi$

The temporal operators \mathbf{G} and \mathbf{F} are of particular importance as they describe *safety* and *liveness* properties respectively. A safety property describes a condition that must always (or never) be met, while a liveness property describes a condition that must eventually be met. Another aspect to consider when checking the temporal properties of a model is

fairness. When modeling a system one may have implicit assumptions about fair behavior, such as requiring that each process in a system be run infinitely often. To handle fairness, a model checking procedure must restrict the state space search to the so-called fair paths of a system, which satisfy the fairness constraints.

In model checking, we are generally concerned with the initialized paths of a model, since they are the possible execution paths starting from an initial state of a system. If all initialized paths of a model M satisfy a property ϕ , we say that $M \models \phi$.

3.2.2 Other Temporal Logics

Another widely used temporal logic is Computation Tree Logic (CTL) [3]. CTL is a branching-time logic, which models time as a tree like structure. In LTL, formulas are implicitly quantified over all paths, so it is not possible to directly check for the existence of a certain path. CTL allows for quantification over paths. In CTL, each of the temporal operators of LTL is combined with a path quantifier **A** or **E**. **A** is the universal path quantifier ('along All paths') and **E** is the existential path quantifier ('there Exists a path'). Certain properties that can be expressed in CTL cannot be expressed in LTL, and vice versa. Neither of the two logics is strictly more expressive than the other.

CTL* [14] and the mu-calculus [40] are temporal logics that subsume both LTL and CTL. As its name suggests, formulas in CTL* are similar to CTL formulas, but combine both state formulas and path formulas. The mu-calculus expresses temporal properties in terms of fixed-point invariants. The mu-calculus provides a single unified logic for temporal properties; LTL, CTL, and CTL* formulas can all be expressed in the mu-calculus. Although these logics are more expressive, they can be more difficult to understand and unintuitive when compared to LTL and CTL. Moreover, the verification procedure for these properties may be more complicated.

3.3 Automata Based Model Checking

In automata based model checking, both the model and the property to be checked are encoded as ω -automata. The class of ω -automata are automata capable of accepting inputs of infinite length. An ω -automaton accepts an infinite input string if the automaton reaches an accepting state infinitely many times. The class of languages (input) ω -automata accept

are the so called ω -regular languages. The simplest ω -automata are Büchi automata. Both Kripke structures and LTL formulas can be represented as Büchi automata.

3.3.1 Büchi Automata

A Büchi automaton is a tuple $A = (\Sigma, S, \rho, S_0, F)$ where:

- Σ is an alphabet,
- S is a set of states,
- $\rho : S \times \Sigma \rightarrow 2^S$ is a nondeterministic transition function,
- $S_0 \subseteq S$ is a set of starting states, and
- $F \subseteq S$ is a set of designated accepting states.

A Kripke structure $M = (S, I, R, L)$ over a set of atomic propositions P can be transformed into a Büchi automaton $A_M = (2^P, S, \rho, I, S)$, where, for $\vec{p} \in 2^P$, $s' \in \rho(s, \vec{p})$ if $(s, s') \in R$ and $L(s') = \vec{p}$.

An algorithm for translating LTL formulas to Büchi Automata was first presented in [54], and the *on-the-fly* technique used by the Spin model checker is presented in [32]. The algorithm computes the states of the automaton by computing the set of subformulas that must hold in each reachable state and in each of its successor states. The initial state is labeled with the full LTL formula and the remainder of the states corresponding to subformulas are computed recursively.

3.3.2 The Model Checking Problem in Terms of Automata

The computation paths of a Kripke structure can be viewed as input words for Büchi automata. This correspondence allows us to view the model checking problem as the problem of computing the intersection of two languages. Let A_M and A_ϕ be Büchi automata corresponding to the model M and property ϕ respectively. The language accepted by A_M , $\mathcal{L}(A_M) \subseteq \Sigma^*$, corresponds to the possible runs of the system, while the language accepted by A_ϕ , $\mathcal{L}(A_\phi)$, correspond to the runs which satisfy the given requirement. If $\mathcal{L}(A_M) \subseteq \mathcal{L}(A_\phi)$, then every run of the system satisfies the property. Let $A_{\neg\phi}$ be the Büchi automata corresponding to $\neg\phi$ (the negation of the property) and $\mathcal{L}(A_{\neg\phi})$ be the language accepted by

that automaton. $\mathcal{L}(A_{\neg\phi})$ corresponds to exactly those runs which violate the property ϕ . The model checking problem can be stated as proving that $\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\phi}) = \emptyset$, i.e. that no run of the system also violates the property being checked. The intersection of the two languages is equivalent to the language accepted by the product automaton $A_M \times A_{\neg\phi}$, which can be computed efficiently. If the intersection of the two languages is not empty, then the intersection provides a counterexample to the property. In summary:

$$\begin{aligned}
M \models \phi & \\
& \text{iff} \\
\mathcal{L}(A_M) \subseteq \mathcal{L}(A_\phi) & \\
& \text{iff} \\
\mathcal{L}(A_M) \cap \mathcal{L}(A_{\neg\phi}) = \emptyset & \\
& \text{iff} \\
\mathcal{L}(A_M \times A_{\neg\phi}) = \emptyset &
\end{aligned}$$

3.3.3 Computing $\mathcal{L}(A_M \times A_{\neg\phi})$

A language $\mathcal{L}(A)$ is not empty if and only if A has an accepting state $s \in F$ that is reachable from an initial state and there is a cycle around s (i.e. there is a path on which $\pi(i) = s$ holds infinitely often for $i \geq 0$). The key problem to be solved then is cycle detection. Spin employs a nested depth first search algorithm to compute $\mathcal{L}(A_M \times A_{\neg\phi})$. A depth first search is first run starting from the initial state to find the reachable accepting states. A second (nested) depth first search is starts from each accepting state to detect cycles. If a cycle is detected then the entire search history can be constructed by concatenating the initial DFS stack with the stack for the current nested search. Spin effectively accomplishes this search *on-the-fly* by running the system automaton and property in alternation (we refer the reader to [32] for details).

3.4 Other Model Checking Approaches

A widely used model checking technique is symbolic model checking [43], which is used by the model checker SMV. In this approach, Kripke structures are represented as boolean functions in the form of ordered binary decision diagrams (BDDs). BDDs allow for efficient manipulation of boolean functions. However, certain functions cannot be represented

compactly as a BDD. Moreover, the size of BDD representation of a function is dependent on variable ordering, and the problem of determining optimal variable ordering for BDDs has been shown to be NP-Complete. Explicit state (automata based) and symbolic (BDD based) model checking techniques each have their strengths and weaknesses. One case study comparing SMV and Spin indicated that, for larger models, Spin produced longer counter examples than SMV. However, Spin was able to find counterexamples faster and required less memory [30]. It is generally held that symbolic model checking is better at verification of true properties, while DFS and automata based approaches are better at producing counterexamples for false properties.

Leveraging the power of state of the art SAT solvers, model checkers based on boolean satisfiability (SAT) have also been proposed [13]. SAT based model checkers are based on the concept of *bounded model checking*, which is another technique used to address the state space explosion problem. In bounded model checking, the goal is not to prove that a system satisfies a property, but to determine if a counterexample of a certain length (upper bound) exists. In this SAT based approach the state transition system, property, and bound are formulated as a boolean propositional formula and a SAT solver is to find a satisfying assignment, which corresponds to a counterexample to the property. However, the bounded model checking method is incomplete, as it can not prove the absence of counterexamples, only their presence.

3.5 Validation Versus Verification

Model checking is generally considered to be a method of formal verification, where the goal is to prove that an implementation conforms to a specification (“Did we build the system right?”). On the other hand, ASMs are geared towards experimental validation of system designs, as they provide an intuitive means of formalizing informal requirements (“Did we build the right system?”). In this sense, one can see that, when an ASM specification is verified with respect to some properties, this verification is part of a larger overall validation process .

Chapter 4

Abstract State Machines

Abstract state machines provide a means of formally specifying systems at a *natural* level of abstraction. This chapter serves as a basic introduction to ASMs. We begin by defining basic ASMs, including definitions of states and standard rules. Then, we define distributed abstract state machines. The chapter finishes with an example ASM specification. For a more detailed and mathematically rigorous definition of ASMs we refer the reader to [11] and [34].

4.1 Basic Abstract State Machines

An abstract state machine is a specific formalization of a state transition system. A basic ASM M is given as $M = (P_M, S_M, I_M)$, where P_M is the program of M , S_M is the set of states associated with M , and $I_M \subseteq S_M$ is the set of initial states of M .

4.1.1 Signatures and States

A *signature* Σ is a finite collection of function names. Each function f , corresponding to a function name in Σ , has some arity $n \geq 0$. The constant nullary function names *undef*, *true*, and *false*, as well as the equality function $=$, are always contained in the signature.

A state $S \in S_M$ is defined as the non-empty *superuniverse* U , which contains elements, along with the interpretations of the function names in Σ over U . Each n -ary function name f in Σ has an interpretation f^S which is a function from U^n to U . The functions *undef*, *true*, and *false* resolve to pairwise distinct elements of U . Functions are total functions and

default to the value of *undef*, which represents an undetermined object. A *relation* is a function that always has the value *true* or *false*, with the default value being *false*.

ASM functions can be divided into several broad categories which have useful semantics to a system modeler. When modeling a (reactive) system it is often beneficial to adopt an open system view, where we view the system as discrete entity operating within some environment. It is useful to distinguish between functions that are under the control of the system (machine) and functions that are controlled by the environment. We say that a function is *controlled* if it can only be changed by the ASM. We say that a function is *monitored* if it can only be changed by the environment. Monitored functions are used to model non-deterministic aspects of a system which are influenced by the environment. Functions whose values can not be changed are called *static* functions. (A static nullary function is also called a *constant*.)

When modeling systems as ASMs, it is useful to subdivide the superuniverse U into smaller universes. One may wish to consider universes as being analogous to types in programming languages. Universes can be described by their characteristic functions. If R is a universe name, the elements which are members of R are exactly those $e \in U$ such that $R^S(e) = \text{true}$. It is possible for an element to be a member of multiple universes.

When designing algorithms we often think in terms of data types such as booleans, integers, sets, strings, etc. Along with the data elements, there is often a set of standard functions defined over them. In ASM, these ideas are captured by the notion of *backgrounds* [6]. A background can be thought of as a static universe of elements which are implicitly part of the state. There may be a set of standard functions defined over those elements as well. For example, the Boolean background consists of the elements *true* and *false* and the standard Boolean operators (\wedge, \vee, \neg , etc).

4.1.2 Locations and Updates

A *location* is a pair $(f, (e_1, \dots, e_n))$, where f is function name and e_1, \dots, e_n are elements from U . The content of location $(f, (e_1, \dots, e_n))$ is the value of $f^S(e_1, \dots, e_n)$ in any state S . Conceptually, locations can be seen as the memory elements of an ASM.

An *update* (l, v) is a pair where l is a location and $v \in U$. Updates change the value of functions – the meaning of an update is to set the content of location l to be the value v . The execution of any ASM rule produces an *update set*, which is a collection of updates. An update set *Updates* is said to be *consistent* if for any location l , if $(l, v) \in \text{Updates}$ and

$(l, w) \in Updates$ then $v = w$. In other words an update set is consistent if it contains no pair of updates which assign different values to the same location. Otherwise, the update set is *inconsistent*. Firing an update set evolves the state of an ASM by applying each update $(l, v) \in Updates$ to the current state. If $Updates$ is inconsistent the update set cannot be fired and there is no state change.

4.1.3 ASM Rules

The program P_M of an ASM M is an ASM rule. The firing (execution) of any ASM rule produces an update set (which may be empty). This section will provide the definition of the standard basic ASM rules. As we will see, rules may be composed from other rules.

Let $\Delta(R)$ denote the update set produced by firing rule R .

- An update rule has the form:

$$f(t_1, \dots, t_n) := t_0$$

where each t_i is a term. Terms are defined recursively as in first-order logic. Variables (nullary functions) are terms, and if g is an n -ary function and t_1, \dots, t_n are terms, then $g(t_1, \dots, t_n)$ is a term as well. The update set produced by the update rule is defined as:

$$\Delta(f(t_1, \dots, t_n) := t_0) = ((f, (v_1, \dots, v_n)), v_0)$$

where v_i is the value of t_i evaluated in the current state. The meaning of this update is that the value of the location $(f, (v_1, \dots, v_n))$ will be v_0 in the next state.

- When describing an algorithm is it often necessary to dynamically allocate new resources by introducing new elements. In ASMs this is accomplished using the **import** rule.

import e **do**
 $R[e]$

where e is a new element imported from the possibly infinite *reserve* universe. Reserve elements can not be referenced by state functions – they can only be accessed through the **import** statement (or extensions of it).

$$\Delta_s(\mathbf{import} \ e \ \mathbf{do} \ R[e]) = \Delta_{s'}(R[e])$$

The element e is removed from the reserve and added to the current state s , resulting in a new state s' . The rule R is evaluated with e in the new state.

- The **skip** rule is a no-op statement, $\Delta(\mathbf{skip}) = \emptyset$.
- A conditional rule has the form:

if $guard$ **then** R_1 **else** R_2

where $guard$ is a Boolean term. The **else** clause may be omitted, implicitly making $R_2 = \mathbf{skip}$.

$$\Delta(\mathbf{if} \ guard \ \mathbf{then} \ R_1 \ \mathbf{else} \ R_2) = \begin{cases} \Delta(R_1) & \text{if the value of } guard \text{ is } true \\ \Delta(R_2) & \text{otherwise} \end{cases}$$

Parallelism

When modeling a system it may be beneficial to abstract away from the order of execution of operations when it is irrelevant to the model. The semantics of ASM allow for rules to be grouped together and executed in parallel. There are two standard ASM rules which allow for the parallel composition of rules, the **par** (block) rule and the **forall** rule.

- The **par** rule has the following form:

par

R_1

R_2

...

R_n

It is often the case that the **par** keyword is omitted and rules listed in sequence are meant to form a parallel block. Each of the R_i 's is executed in parallel, so

$$\Delta(\mathbf{par} R_1 \dots R_n) = \Delta(R_1) \cup \dots \cup \Delta(R_n)$$

- The **forall** rule allows the same rule to be executed simultaneously on each of the members of a collection. It has the form

forall $e \in C$ **with** *guard* **do**
 $R[e]$

where C is an enumerable collection. The rule R is executed on each member of the collection that satisfies the guard condition. When the **with** clause is omitted this implicitly means that *guard* = *true*.

$$\Delta(\mathbf{forall} e \in C \mathbf{with} \textit{guard} \mathbf{do} R[e]) = \Delta(R[c_1]) \cup \dots \cup \Delta(R[c_n])$$

where $c_i \in \{ c \mid c \in C, \textit{guard}[c] = \textit{true} \}$.

Non-determinism

Non-determinism can be useful in describing a program behavior at a high level of abstraction. Non-deterministic constructs can be used when selection decisions, such as process scheduling, are arbitrary, and to model random processes.

- ASMs can express non-determinism through the **choose** rule. It has the form

choose $e \in C$ **with** *guard* **do**
 $R_1[e]$
ifnone
 R_2

where C is an enumerable collection. When the **with** clause is omitted this implicitly means that *guard* = *true*, and when the **ifnone** clause is omitted this implicitly means that $R_2 = \mathbf{skip}$.

$$\Delta(\mathbf{choose} \ e \in C \ \mathbf{with} \ guard \ \mathbf{do} \ R_1[e] \ \mathbf{ifnone} \ R_2) = \begin{cases} \Delta(R_1[c]) & \text{if } C_g \neq \emptyset \\ \Delta(R_2) & \text{otherwise} \end{cases}$$

where c is chosen non-deterministically from $C_g = \{ c \mid c \in C, guard[c] = true \}$.

Supplementary Rule Forms

- An **extend** rule has the form:

extend U **with** e **do**
 $R[e]$

where U is a universe name. The **extend** rule is an extension of the **import** rule.

$$\Delta_s(\mathbf{extend} \ U \ \mathbf{with} \ e \ \mathbf{do} \ R[e]) = \{((U, (e)), true)\} \cup \Delta_{s'}(R[e])$$

Like the **import** rule, **extend** imports a new element from the reserve and adds it to the current state s . In addition to this, the new element is added to the universe U ($U(e) = true$). The rule R is then evaluated in the new state s' .

- For the purpose of reuse and modularization, it is possible to declare named parameterized rules, with $n \geq 0$ parameters:

$rulename(x_1, \dots, x_n) =$
 $R[x_1, \dots, x_n]$

A call rule has the form:

$rulename(t_1, \dots, t_n)$

where $rulename$ is the name of a declared rule and t_1, \dots, t_n are arbitrary terms. The meaning of the call rule is to substitute each x_i with the corresponding t_i and fire rule R .

$$\Delta(rulename(t_1, \dots, t_n)) = \Delta(R[t_1, \dots, t_n])$$

4.2 Distributed Abstract State Machines

A *distributed abstract state machine* (DASM) M has a finite universe of *Agents* and, exactly as in a basic ASM, a set of states S_M and a set of initial states I_M . Each agent $a \in \text{Agents}$ has an associated program that defines its behavior, which is defined by the value of the location $\text{program}(a)$, where program is a dynamic function. The static collection of programs that the agents may execute make up the distributed program P_M . The vocabulary of a distributed abstract state machine also has a special static function self , which is interpreted differently by each agent to refer to itself. For an agent a , $\text{self}^S = a$.

A run ρ of a DASM M [34], also called a partially ordered run, can be defined as a triple (P, A, σ) satisfying the following conditions:

1. P is a partially ordered set of moves, where each move has finitely many predecessors, i.e. $\{y \mid y \leq x\}$ is finite.
2. The function A associates a move in P with the agent that performs that move. ($A(x)$ is the agent performing move x .) The moves of any single agent are linearly ordered, so every nonempty set $\{x \mid A(x) = a\}$ is linearly ordered.
3. $\sigma(X)$ is the state of M produced by performing all the moves in X . σ is defined over all the initial segments P and on the empty set; $\sigma(\emptyset)$ is an initial state.
4. The coherence condition: If x is a maximal element in a finite initial segment X of P and $Y = X - \{x\}$, then $A(x)$ is an agent in $\sigma(Y)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $A(x)$ at $\sigma(Y)$.

An immediate and useful corollary of the coherence condition is that all linearizations of an initial segment of a run result in the same final state. While the above definition is concise, understanding of the coherence condition and its implications will benefit from some illustration.

Light Control Example (Adapted from [10])

The light control system for a building is modeled as a DASM. The model has three distinct agents: the window manager (agent W), the door manager (agent D), and the light manager (agent L). The state of the system is represented by the nullary Boolean functions *window*, *door*, and *light*. When the value of each of these functions is *true*, it has the

intuitive meaning of “the window is open”, “the door is open” and “the lights are turned on” respectively. The behavior of the agents is described by the following rules:

- $WindowManagerProgram = \mathbf{if} \neg door \mathbf{then} window := true$
- $DoorManagerProgram = \mathbf{if} \neg window \mathbf{then} door := true$
- $LightManagerProgram = \mathbf{if} \neg door \vee \neg window \mathbf{then} light := true$

Let w , d , and l be the moves performed by each of these agents respectively. Figure 4.1 illustrates all possible runs of the system starting from the initial state S_0 , where the door and window are closed and the lights are off. According to the definition of partially ordered runs the move from S_0 to S_6 is not permissible, since the two possible linearizations of the moves $\{d, w\}$ do not result in the same final state. If move d is made first, resulting in state S_1 , W can not make its move to S_6 . Similarly, if move w is made first, resulting in state S_3 , D can not make its move to S_6 . By the exact same reasoning, the move from S_0 to S_7 is also not permissible. On the other hand, notice that the moves in $\{d, l\}$ and in $\{l, w\}$ produce the same resultant states, S_4 and S_5 respectively, regardless of the ordering of the individual moves.

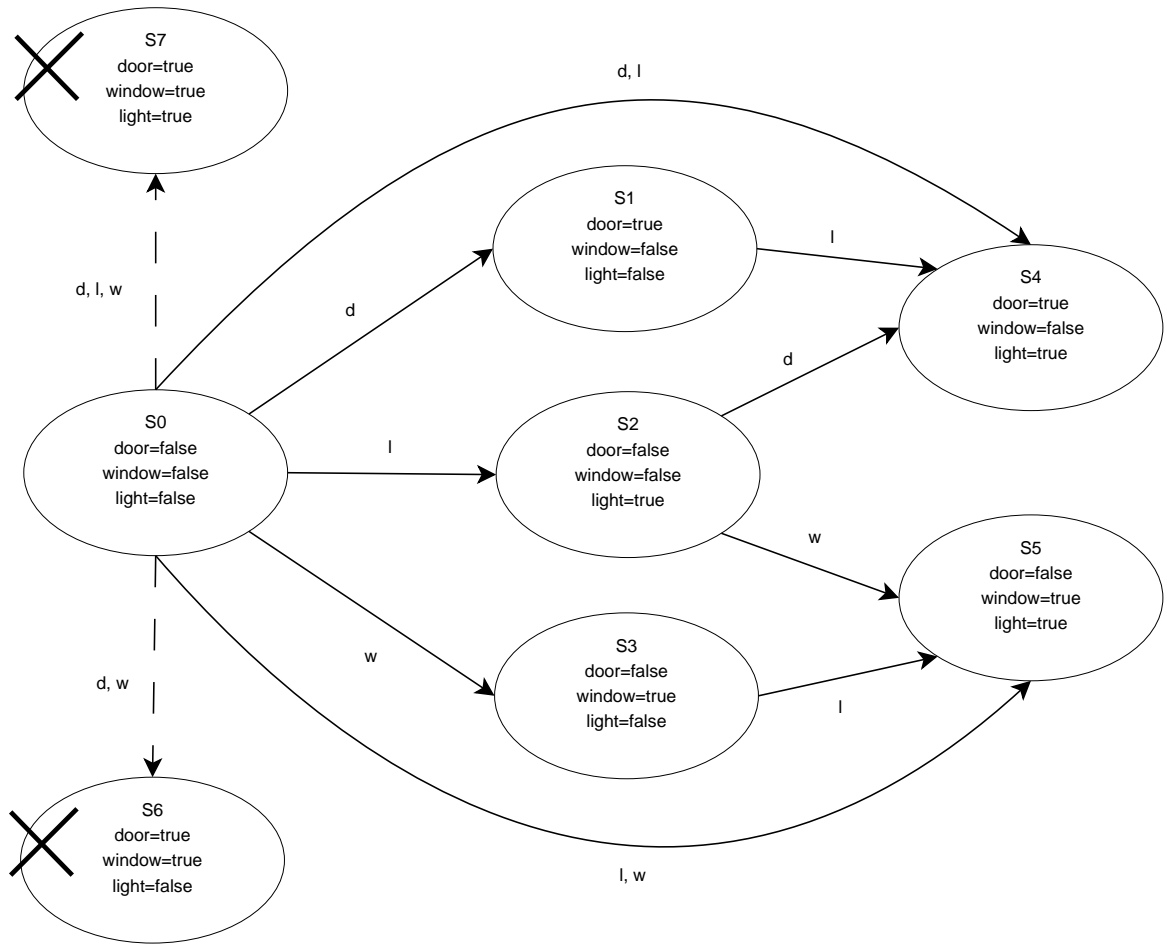


Figure 4.1: Partially Ordered Runs of the Light Control System

4.3 A Sample ASM Specification: Master-slave Agreement

The system being modeled consists of one master process and a number of uniform slave processes. The problem is for the master to delegate a job to the slave processes. Before a job can be delegated to the slave processes, the master must query the slaves to see if they are free to accept the job. If all of the slave processes accept then the job is executed. Otherwise, the job is canceled.

The signature of the DASM model for this system is as follows:

	Master-Slave Agreement: Signature
$Slave = \{s_1, \dots, s_n\}$ $Agents = Slave \cup \{master\}$	
$MODE = \{idle, waitingForAnswer, waitingForOrder, busy\}$ $REPLY = \{accept, refuse\}$ $ORDER = \{job, cancel\}$	
$controlled\ asked : Slave \rightarrow BOOLEAN$ $controlled\ answer : Slave \rightarrow REPLY$ $controlled\ order : \rightarrow ORDER$ $controlled\ mode : Agent \rightarrow MODE$	

In this signature, $MODE$ is the background defining the possible modes of an *Agent*, $REPLY$ is the background defining slave responses, and $ORDER$ is the background defining the orders that can be given by the master process. The function *asked* indicates if a slave is being queried by the master process; its value is initially *false* for all slaves. The function *answer* is the response given by a slave process; its value is initially *undef* for all slaves. The function *order* is the order given by the master process; its value is initially *undef*. Finally, the function *mode* gives the current mode of an agent; its value is initially *undef* for all agents.

The master process' program is given by the rule *MasterProgram*. The master process queries the slave processes, asking each slave if it can accept a job. The master process waits for all the slaves to respond and if all the slaves are able to accept a job the master issues the job order, otherwise the master process cancels the job request.

```

MasterProgram  $\equiv$ 
  if  $mode(self) = idle$  then
    forall  $s \in Slave$  do
       $asked(s) := true$ 
       $mode(self) := waitingForAnswer$ 
    if  $mode(self) = waitingForAnswer \wedge \forall s \in Slave \neg(answer(s) = undef)$  then
       $OrderOrCancel$ 
       $mode(self) := idle$ 

```

```

OrderOrCancel  $\equiv$ 
  if  $\exists s \in Slave$  with  $answer(s) = refuse$  then
     $order := cancel$ 
  else
     $order := job$ 
  forall  $s \in Slave$  do
     $answer(s) := undef$ 

```

The program of the slave processes are given by the rule *SlaveProgram*. A slave process waits to be queried by the master. After being queried, a slave non-deterministically chooses a response. After responding, a slave waits to receive and order from the master.

SlaveProgram \equiv

```
if  $mode(self) = idle \wedge asked(self)$  then
  SendAnswer
if  $mode(self) = waitingForOrder$  then
  if  $order = job$  then
     $mode(self) = busy$ 
  if  $order = cancel$  then
     $mode(self) = idle$ 
     $order = undef$ 
```

SendAnswer \equiv

```
choose  $r \in REPLY$  do
   $answer(self) := r$ 
   $asked(self) := false$ 
   $mode(self) := waitingForOrder$ 
```

Chapter 5

Extending CoreASM for Model Checking

Since unbounded model checking performs a full state space search of a transition system, a system must be finite for unbounded model checking to be possible. Thus to model check an ASM, its functions and universes, which make up its state, must be finite. However, in principle, ASM functions are untyped alike functions; alternately, one can view the arguments and values of ASM functions as all being of the same type – they are all elements from the superuniverse. As *CoreASM* follows the mathematical definition of ASMs, *CoreASM* functions are untyped at the base (kernel) level. While this is desirable in initial specification phases focusing on exploring the problem space, the domain and range types of functions must be known and finite for model checking of *CoreASM* specifications.

This chapter describes the Signature Plug-in which extends the *CoreASM* language to include function declarations with type information. Moreover, when performing model checking, one must of course specify a property to be checked. It is convenient to include this property as part of the specification. This chapter also describes the Property Plug-in, which extends the *CoreASM* language to include correctness properties that are expressed as LTL formulas. Before presenting these two plug-ins, we first give an overview of the architecture of *CoreASM*, with a focus on the components that are relevant to this thesis. Since our assertion is that the ASM method is well suited to modeling arbitrary software systems, it is only fitting that we specify the *CoreASM* engine, language semantics, and plug-ins using ASM. (In fact, the semantics of several well known computer languages, including

SDL [49], VHDL [8], Java [52], and C# [7], have been specified using the ASM formalism.)

5.1 CoreASM: An Overview

The material in this section borrows from work originally presented by Farahbod et al. [24, 23, 25, 27, 26]. The CoreASM project [22] focuses on the design of a lean executable ASM language, in combination with a supporting tool environment for high-level design, experimental validation and, where appropriate, formal verification of abstract system models. CoreASM is designed with extensibility in mind, supporting the extension of both the specification language and the execution engine's behavior through *plug-ins*.

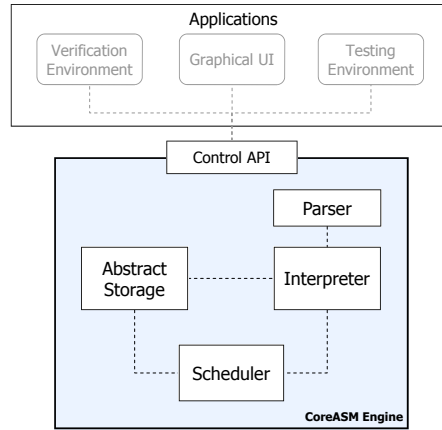
The CoreASM engine is composed of four components: the *Parser*, the *Interpreter*, the *Abstract Storage*, and the *Scheduler* (see Figure 5.1¹). When a specification is executed, the Parser first dynamically generates a parser for the specification, whose language grammar depends on the plug-ins that are used by the specification. This custom parser is used to parse the specification. Parsing of a specification produces an *Abstract Syntax Tree* (AST). The rules represented in the AST are executed by the Interpreter, producing updates. The Interpreter interacts with the Abstract Storage and the Scheduler to apply these updates thereby evolving the state of the simulated ASM. The Abstract Storage maintains a representation of the ASM state. The Scheduler schedules agents to be run, and coordinates the overall execution of ASM runs.

The CoreASM engine adopts a micro-kernel architecture. The base machine (kernel) only supports two basic ASM rules, *assignment* and *import*. This is the minimal set of required rules, since without assignment there would be no means of evolving the state, and without import new elements could not be introduced into the state. The kernel also contains the special element *undef*, and the elements from the Boolean background, *true* and *false*, since universes are defined by their characteristic functions. Other rule forms (such as conditional, forall, and extend) and backgrounds are introduced through plug-ins, which extend conservatively from the kernel (see Figure 5.2).

The overall process of executing a specification with the CoreASM engine consists of three macrosteps, each of which includes a number of microsteps as follows:

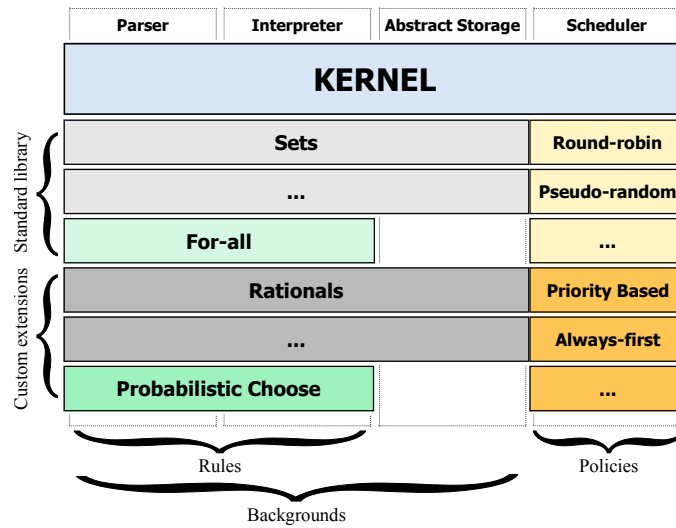
1. Initializing the engine (Figure 5.3)

¹Figures 5.1 - 5.6 are taken from [26] by permission.



©Roozbeh Farahbod, 2006, by permission

Figure 5.1: Overall Architecture of CoreASM

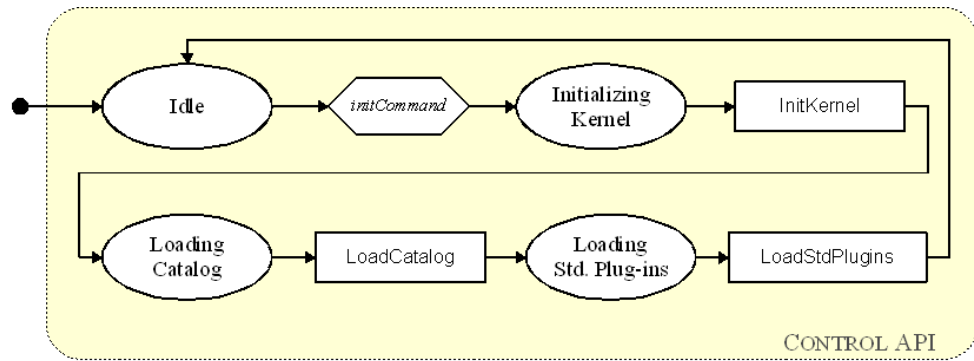


©Roozbeh Farahbod, 2006, by permission

Figure 5.2: Layers and Modules of the CoreASM Engine

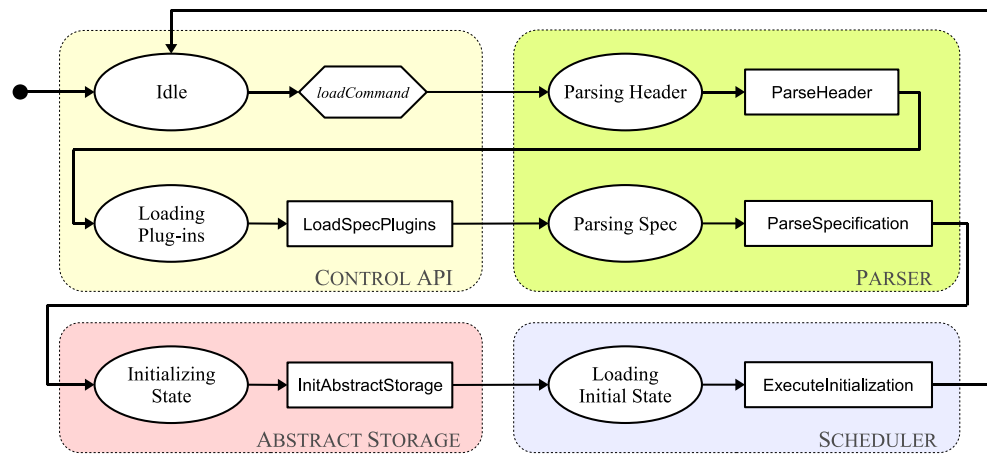
- (a) Initializing the kernel
 - (b) Loading the plug-in library catalog
 - (c) Loading and activating the essential (core) plug-ins
2. Loading a CoreASM specification (Figure 5.4)
 - (a) Parsing the specification header
 - (b) Loading further needed plug-ins as declared in the header
 - (c) Parsing the specification body
 - (d) Initializing the abstract storage
 - (e) Setting up the initial state
 3. Execution of the specification (Figure 5.5)
 - (a) Execute a single step
 - (b) If termination condition not met, repeat from 3a

Figures 5.3-5.5 are Control State ASM diagrams which outline the operation of the engine. The lower level details of the CoreASM engine are not required to understand the work presented in this thesis. We refer the reader to [24] for a much more detailed specification of the CoreASM engine.



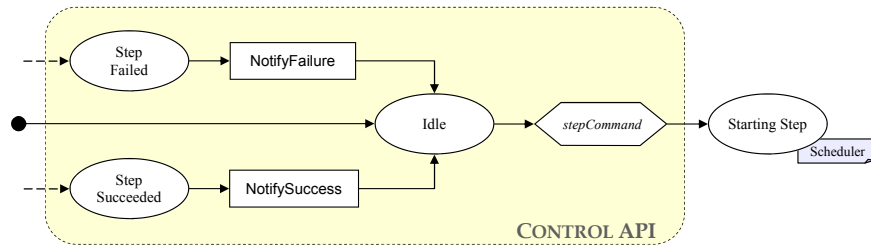
©Roozbeh Farahbod, 2006, by permission

Figure 5.3: Control State ASM of Initializing CoreASM Engine

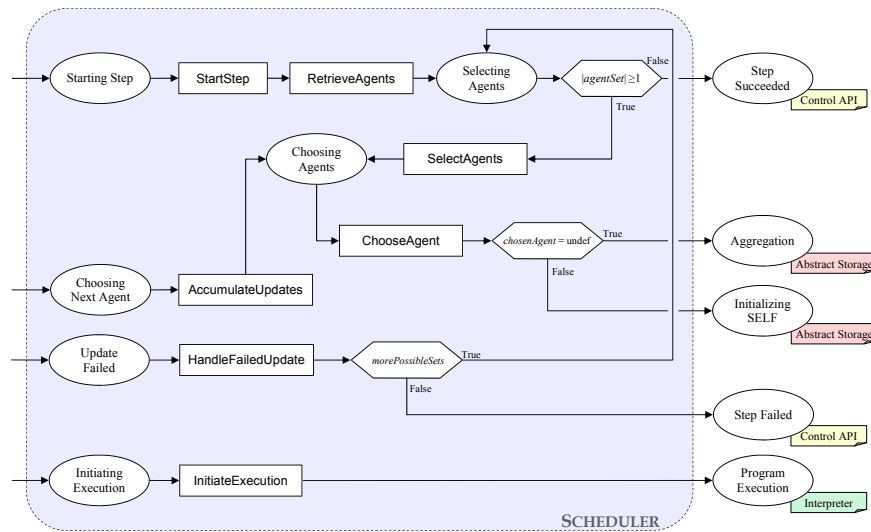


©Roozbeh Farahbod, 2006, by permission

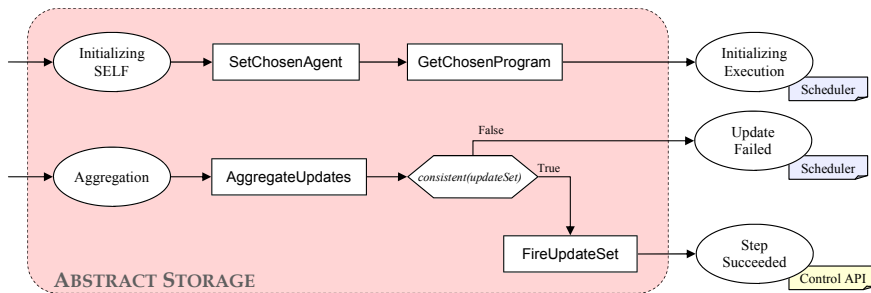
Figure 5.4: Control State ASM of Loading a CoreASM Specification



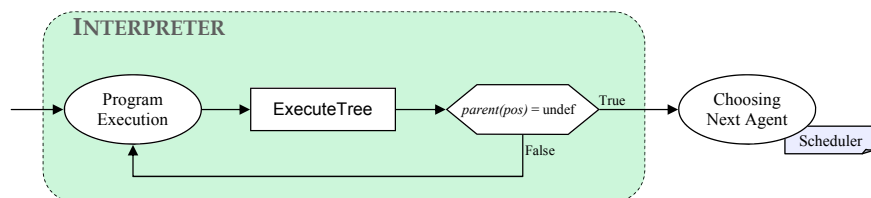
(a) Control API Module



(b) Scheduler



(c) Abstract Storage



(d) Interpreter

©Roosbeh Farahbod, 2006, by permission

Figure 5.5: Control State ASM of a STEP command

Plug-in Interface	Extends	Description
<i>Parser Plug-in</i>	Parser	provides additional grammar rules to the parser
<i>Interpreter Plug-in</i>	Interpreter	provides new semantics to the interpreter
<i>Operator Provider</i>	Parser, Interpreter	provides grammar rules for new operators along with their precedence levels and semantics
<i>Vocabulary Extender</i>	Abstract Storage	extends the state with additional functions, universes, and backgrounds
<i>Aggregator</i>	Abstract Storage	aggregates partial updates into basic updates
<i>Scheduler Plug-in</i>	Scheduler	provides new scheduling policies for multi-agent ASMs
<i>Extension Point Plug-in</i>	all components	extends the control state model of the engine

Table 5.1: CoreASM Plug-in Interfaces

5.1.1 CoreASM Plug-in Framework

Concretely, a CoreASM plug-in is a Java² class that inherits from the base Plug-in class and implements one or more of CoreASM's plug-in interfaces (see Table 5.1). With these interfaces a plug-in can extend the engine components and the control state model of the engine. We will now describe these interfaces.

Parser Extensions

Plug-ins can extend the parser by implementing the *Parser Plug-in* interface and/or the *Operator Provider* interface. These interfaces allow a plug-in to contribute new grammar rules and operators to the parser respectively. For any parser plug-in pp , $pluginGrammar(pp)$ holds the set of all the grammar rules contributed by pp . For any operator provider op , $pluginOperators(op)$ holds the descriptions (syntax) of new operators contributed by op .

Before parsing a specification, the engine gathers all the grammar rules and operator descriptions provided by all parser plug-ins and operator providers. These grammar rules and operator tokens are then combined with the kernel grammar to build a new custom 'parser' to parse the specification. While building the abstract syntax tree, this parser labels

²The CoreASM Engine is implemented in Java.

the nodes that are created by plug-in-provided grammar rules with the plug-in identifier; these labels can later be used by the interpreter to evaluate such nodes.

Interpreter Extensions

By implementing the *Interpreter Plug-in* interface and/or the *Operator Provider* interface, plug-ins can extend the interpreter component of the engine. These plug-ins provide the semantics of rules and operations. Traversing the abstract syntax tree, the `ExecuteTree` rule of the interpreter (see Figure 5.5(d)) uses these semantic rules to evaluate nodes that correspond to the extended grammar rules.

The semantics contributed by a plug-in p which implements the *Interpreter Plug-in* interface can be obtained through $pluginRule(p)$. As already mentioned earlier, nodes of the parse tree corresponding to grammar rules provided by a plug-in are annotated with the plug-in identifier. If a node refers to a plug-in, the interpreter obtains the semantic rules provided by that plug-in and executes it; otherwise, the default kernel interpreter rules are used.

The `ExecuteTree` rule of the interpreter is presented below. In this rule, the current position in the abstract syntax tree is denoted by the nullary function pos , and assignment to pos is used to move evaluation to a different node. We refer the reader to [23, 27] for more details on this process.

Interpreter

```

ExecuteTree  $\equiv$ 
  if  $\neg evaluated(pos)$  then
    if  $plugin(pos) \neq undef$  then
      let  $R = pluginRule(plugin(pos))$  in
         $R$ 
    else
      KernellInterpreter
  else
    if  $parent(pos) \neq undef$  then
       $pos := parent(pos)$ 

```

A similar approach is also used by the `KernellInterpreter` rule to obtain semantics of extended operators from *Operator Providers*. A detailed discussion on how the engine deals with operators and their extensions is provided in [44].

Abstract Storage Extensions

Vocabulary Extender plug-ins can extend the vocabulary of the CoreASM state by contributing new backgrounds, universes, and functions to the abstract storage. Such plug-ins in fact extend the initial state and signature of the simulated ASM.

In the abstract storage, the following functions bind the names of functions and universes in the CoreASM state to the mathematical objects that represent them. Backgrounds are considered as special universes and hence are handled by the same mapping.

- $stateUniverse : STATE \times NAME \rightarrow UNIVERSEELEMENT$
- $stateFunction : STATE \times NAME \rightarrow FUNCTIONELEMENT$

The value of these functions is initialized by the `InitAbstractStorage` rule of the abstract storage (see Figure 5.4). After creating the default universe and functions (i.e., “Agents”, “program”, and “self”), this rule iterates over all vocabulary extender plug-ins and extends the CoreASM state with the vocabulary they provide:

Abstract Storage

InitAbstractStorage \equiv

InitializeState

forall $p \in specPlugins$ **do**

if $isVocabularyExtender(p)$ **then**

forall $(bkgName, bkg) \in pluginBackgrounds(p)$ **do**

$stateUniverse(state, bkgName) := bkg$

forall $(uName, universe) \in pluginUniverses(p)$ **do**

$stateUniverse(state, uName) := universe$

forall $(fName, f) \in pluginFunctions(p)$ **do**

$stateFunction(state, fName) := f$

Plug-ins can also implement the *Aggregator* interface and provide aggregation rules to be applied on update instructions before they are submitted to the state. Aggregators are used, for example, to implement partial updates; for more detail on this issue, we refer the reader to [44].

Scheduler Extensions

Policy plug-ins extend the scheduler of the engine by providing new scheduling policies that affect the selection of agents in multi-agent ASMs. They provide an extension to the scheduler that is used to determine at each step the next set of agents to execute. In practice, a scheduler plug-in provides a concrete implementation of a **choose** in the **SelectAgents** step in Figure 5.5(b). It is worthwhile to note that only a single scheduling policy can be in force at any given time, whereas an arbitrary number of plug-ins of the remaining types can be all in use at the same time.

Extension Point Plug-ins

In addition to modular extensions of specific components, plug-ins can also extend the control state of the engine by registering themselves for *Extension Points*. Each mode transition in the execution engine is associated to an extension point. At any extension point, if there is any plug-in registered for that point, the code contributed by the plug-in for that transition is executed before the engine proceeds into the new mode. Such a mechanism enables arbitrary extensions to the engine’s life-cycle, which facilitates implementing various practically relevant features such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the behavior of the simulated machine (e.g., coverage analysis or profiling).

5.1.2 State Representation in CoreASM

Elements

The base data elements used by the Abstract Storage to represent an ASM state are simply referred to as **ELEMENTS**. All other elements of the state, including functions, universes, backgrounds, and rules, extend from **ELEMENTS** (see Figure 5.6). The following functions are defined over all elements of the state:

- $bkg : \text{ELEMENT} \rightarrow \text{NAME}$
is the name of the background of the given element. The default value is “Element”.
- $equal_{Element} : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$

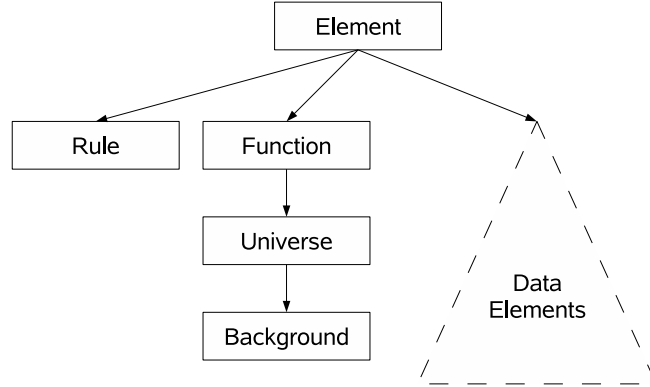


Figure 5.6: Core Elements in the Kernel

returns *true* if the two elements are equal. We have

$$\forall a_1, a_2 \in \text{ELEMENT} \quad \neg \text{equal}_{\text{Element}}(a_1, a_2)$$

- **derived** $\text{equal} : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$

returns *true* if the given elements are equal. This function is defined as

$$\text{equal}(a_1, a_2) \equiv \text{equal}_{\text{bkg}(a_1)}(a_1, a_2) \vee \text{equal}_{\text{bkg}(a_2)}(a_2, a_1)$$

Enumerable Elements

An element is *enumerable* if it can be viewed as a collection (i.e., a multiset) of other elements. The idea of enumerable elements provides a unique and yet simple interface to sets, multisets, trees, and other data structures. We define the following functions to support enumerable elements:

- **controlled** $\text{enumerable} : \text{ELEMENT} \rightarrow \text{BOOLEAN}$

holds *true* if the element is enumerable.

- **derived** $\text{enumerate} : \text{ELEMENT} \rightarrow \text{ELEMENT-COLLECTION}$

provides a collection of elements representing the internal structure of the enumerable element.

$$\text{enumerate}(e) \equiv \text{enumerate}_{\text{bkg}(e)}(e)$$

- **derived** $contains : \text{ELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$

holds *true* if and only if the first element is enumerable and as a collection it contains the second element.

$$contains(e_1, e_2) \equiv \begin{cases} true, & \text{if } enumerable(e_1) = true \wedge e_2 \in enumerate(e_1) \\ false, & \text{otherwise.} \end{cases}$$

Function Elements

Function elements represent the functions that are defined in a CoreASM state.

FUNCTIONELEMENT, a subset of ELEMENT, is introduced to provide a core concept for state functions (tables) and custom-defined functions (e.g., derived functions provided by a plugin, such as ‘*sin(x)*’). The following functions and rule are defined over functions elements:

- $fClass : \text{FUNCTIONELEMENT} \rightarrow \text{FUNCCLASS}$

Is the class of the function, where

$$\text{FUNCCLASS} \equiv \{ monitored, controlled, out, static, derived \}$$

The default value of this function is *controlled*.

- $fGetValue : \text{FUNCTIONELEMENT} \times \text{ELEMENT-SEQ} \rightarrow \text{ELEMENT}$

returns the value of this function with respect to the given arguments. The default value of this function is *undef*.

- **rule** $FSetValue(f, args, v)$

sets a new value for the function, if this is possible. By default, this rule is defined as follows.

$$\begin{aligned} \mathbf{FSetValue}(f, args, v) &\equiv \\ fGetValue(f, args) &:= v \end{aligned}$$

FSetValue

- $signature : \text{FUNCTIONELEMENT} \rightarrow \text{SIGNATURE}$

is the signature of the given function. The default value of this function is *undef*.

- **derived** $fLocations : \text{FUNCTIONELEMENT} \rightarrow \text{LOC-SET}$

is the set of all locations for which this function has a value other than *undef*.

- **derived** $equal_{Function} : FUNCTIONELEMENT \times ELEMENT \rightarrow BOOLEAN$
where we have,

$$equal_{Function}(f_1, f_2) \equiv \forall a \in ELEMENT-SEQ \ fGetValue(f_1, a) = fGetValue(f_2, a)$$

- $\forall f \in FUNCTIONELEMENT \ bkg(f) = \text{"Function"}$

Locations

Locations within a state are pairs of function names and arguments lists.

- $locName : LOC \rightarrow NAME$
is the name of the function on which this location is defined.
- $locArgs : LOC \rightarrow ELEMENT-SEQ$
is the list of abstract object values, as arguments of the location.
- **derived** $locFunction : LOC \rightarrow FUNCTIONELEMENT$
is the function on which this location is defined.

$$locFunction(l) = f \Leftrightarrow name(f) = locName(l)$$

Signature Elements

Signature elements represent the signatures of functions. Domain and range types (universes) are identified by their names. SIGNATURE elements have the following functions defined on them:

- **controlled** $sigDomain : SIGNATURE \rightarrow NAME-SEQ$
is the ordered list of domain types for this signature.
- **controlled** $sigRange : SIGNATURE \rightarrow NAME$
is the range type for this signature.

Universe Element

Universe elements represent the universes that are defined in a CoreASM state. Hence, UNIVERSEELEMENT is a subset of FUNCTIONELEMENT. The following functions are defined on these elements:

- $uMember : \text{UNIVERSEELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
is the membership function of the universe. If u is a universe, then we may use the syntactical form $u(x)$ for $uMember(u, x)$.
- **derived** $equal_{Universe} : \text{UNIVERSEELEMENT} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
where

$$equal_{Universe}(a, b) = equal_{Function}(a, b)$$
- $\forall i \in \text{UNIVERSEELEMENT} \quad bkg(i) = \text{"Universe"}$

For all $v \in \text{ELEMENT}$ and $u \in \text{UNIVERSEELEMENT}$, we have,

- $fGetValue(u, v) \equiv uMember(u, v)$
- $FSetValue(u, \langle v \rangle, b) \equiv uMember(u, v) := b$

Background Elements

Background elements represent backgrounds (static universes) in a CoreASM state.

BACKGROUND is a subset of UNIVERSEELEMENT.

- **controlled** $newValue : \text{BACKGROUND} \rightarrow \text{ELEMENT}$
returns a pseudo new element of the given background; i.e., most probably returns a default value like an empty string for strings, or an empty set for sets, or *false* for Booleans.
- $\forall b \in \text{BACKGROUND} \quad fClass(b) = \text{static}$
It is not possible to change the membership function of a background; i.e., it is not possible to add any element to a background or to remove any element from it.
- **derived** $equal_{Background} : \text{BACKGROUND} \times \text{ELEMENT} \rightarrow \text{BOOLEAN}$
where

$$equal_{Background}(a, b) = equal_{Universe}(a, b)$$
- $\forall i \in \text{BACKGROUND} \quad bkg(i) = \text{"Background"}$

Rule Elements

Rule elements represent the ASM rules that are defined in a CoreASM state. Thus, RULE is a subset of ELEMENT. The following functions are defined on rule elements:

- $ruleName : RULE \rightarrow NAME$
is the name of the rule. If not *undef*, this name must be unique in state.
- $body : RULE \rightarrow NODE$
holds the body (syntax tree) of the rule.
- $param : RULE \rightarrow TOKEN-SEQ$
holds (in order) the parameters of the rule in squence of tokens (or strings).
- **derived** $equal_{Rule} : RULE \times ELEMENT \rightarrow BOOLEAN$
where

$$equal_{Rule}(a, b) = equal_{Element}(a, b)$$
- $\forall i \in RULE \quad bkg(i) = \text{"Rule"}$

State

The state of a simulated machine is represented as an abstract data structure. The following functions define the interface of such a data structure:

- $content : STATE \times LOC \rightarrow ELEMENT$
is the value of a given location in the state. This function represents the interface of the state.
- $stateUniverse : STATE \times NAME \rightarrow UNIVERSEELEMENT$
is the mapping of universe names to universe elements in the state.
- $stateFunction : STATE \times NAME \rightarrow FUNCTIONELEMENT$
is the mapping of function names to function elements in the state.
- $stateRule : STATE \times NAME \rightarrow RULE$
is the mapping of rule names to rule elements in the state.

The following derived functions are defined to respectively provide the set of all the universes, functions, rules, and locations defined in a state.

- **derived** $universes : \text{STATE} \rightarrow \text{UNIVERSEELEMENT-SET}$

$$universes(s) \equiv \{u \mid u \in \text{UNIVERSEELEMENT} \wedge (\exists n \in \text{NAME}, \text{stateUniverse}(s, n) = u)\}$$

- **derived** $functions : \text{STATE} \rightarrow \text{FUNCTIONELEMENT-SET}$

$$functions(s) \equiv \{f \mid f \in \text{FUNCTIONELEMENT} \wedge (\exists n \in \text{NAME}, \text{stateFunction}(s, n) = f)\}$$

- **derived** $rules : \text{STATE} \rightarrow \text{RULE-SET}$

$$rules(s) \equiv \{r \mid r \in \text{RULE} \wedge (\exists n \in \text{NAME}, \text{stateRule}(s, n) = r)\}$$

- **derived** $locations : \text{STATE} \rightarrow \text{LOC-SET}$

$$locations(s) \equiv \{l \mid \exists f (f \in functions(s) \wedge l \in fLocations(f))\}$$

- **derived** $isUniverseName : \text{NAME} \rightarrow \text{BOOLEAN}$

$$isUniverseName(name) \equiv universes(state, name) \neq \text{undef}$$

- **derived** $isFunctionName : \text{NAME} \rightarrow \text{BOOLEAN}$

$$isFunctionName(name) \equiv functions(state, name) \neq \text{undef}$$

- **derived** $isRuleName : \text{NAME} \rightarrow \text{BOOLEAN}$

$$isRuleName(name) \equiv rules(state, name) \neq \text{undef}$$

5.2 Signature Plug-in

In principle, CoreASM functions are untyped alike ASM functions. While this is desirable in initial specification phases focusing on exploring the problem space, domain and range types of functions often add useful semantic information to a refined specification, for instance,

to improve its understandability, to implement runtime type checking, and also to facilitate model checking. The Signature Plug-in provides means to declare functions with their associated signatures, thereby adding type information to CoreASM. Moreover, it also allows to define new universes and enumerated backgrounds directly in a specification, rather than introducing them by a separate plug-in.

The Signature Plug-in extends the parser, the interpreter and the abstract storage. Extending the grammar of the CoreASM language with its own syntactic patterns, the Signature plug-in creates new nodes in the AST. These nodes are not evaluated during the execution of the ASM, since they do not represent regular rules or expressions; rather they are interpreted before an ASM run, when the engine is in the *Initializing State* mode (see Figure 5.4). During the initialization of the abstract storage, the engine queries plug-ins for the vocabulary elements they provide (see definition of `InitAbstractStorage` in Section 5.1.1). Hence, the interpretation of Signature declarations directly modifies the initial state (and vocabulary of the machine).

5.2.1 Functions

To declare functions, the Signature plug-in extends the CoreASM language with the following syntactic patterns³, which can appear in the header of a specification:

	Function Declaration
$(\mathbf{function} \ x_{name} : x_{d_1} * \dots * x_{d_n} \rightarrow x_r)$	\rightarrow <code>createFunction(x_{name}, <i>controlled</i>, $\langle x_{d_1}, \dots, x_{d_n} \rangle$, x_r)</code>
$(\mathbf{function \ controlled} \ x_{name} : x_{d_1} * \dots * x_{d_n} \rightarrow x_r)$	\rightarrow <code>createFunction(x_{name}, <i>controlled</i>, $\langle x_{d_1}, \dots, x_{d_n} \rangle$, x_r)</code>
$(\mathbf{function \ static} \ x_{name} : x_{d_1} * \dots * x_{d_n} \rightarrow x_r)$	\rightarrow <code>createFunction(x_{name}, <i>static</i>, $\langle x_{d_1}, \dots, x_{d_n} \rangle$, x_r)</code>
$(\mathbf{function \ monitored} \ x_{name} : x_{d_1} * \dots * x_{d_n} \rightarrow x_r)$	\rightarrow skip

Although (at the time of writing) monitored functions are not supported in the CoreASM interpreter, the syntactic pattern for declaring monitored functions is included above

³The notation we use here has been borrowed from [27]. It will suffice to say that the semantics is given by ASM rules guarded by syntactical patterns. Patterns are delimited by $(\) \rightarrow$ symbols; inside a pattern, variables named x , e , v indicate that the corresponding node or subtree is an identifier, an expression, a value. An empty box indicates an unevaluated node; a boxed letter indicates an unevaluated node which is expected to result in the corresponding element. superscripts name locations.the corresponding value in the pattern.

because monitored functions are supported in the translation of CoreASM to Promela. The interpretation of function declaration patterns is defined by the `createFunction` rule, which creates a new function and with the specified name, class, and signature.

createFunction

```

createFunction(name, functionClass, domain, range) ≡
  let f = new(FUNCTIONELEMENT) in
    fClass(f) := functionClass
  let s = new(SIGNATURE) in
    sigDomain(s) := domain
    sigRange(s) := range
    signature(f) := s
  add (name, f) to pluginFunctions(SignaturePlugin)

```

One can also specify the initial value(s) of a function in the function declaration by including an initialization expression at the end of the declaration. The initialization expression may be a basic expression, for nullary functions, or a map expression, for n -ary functions. Before the function is created, the expression giving its initial value is evaluated. In the following rule *fClass* is either of *static* or *controlled*.

	Function Declaration with Initialization
(function <i>fClass</i> $x_{d_1} * \dots * x_{d_n} \rightarrow x_r$ initially $\alpha \boxed{e}$)	→ <i>evaluate</i> (α)
(function <i>fClass</i> $x_{d_1} * \dots * x_{d_n} \rightarrow x_r$ initially αv)	→
	createFunction(x_{name} , <i>fClass</i> , $\langle x_{d_1}, \dots, x_{d_n} \rangle$, x_r , v)

To support function value initialization the `createFunction` rule is modified as follows:

createFunction with Initial Value

```

createFunction(name, functionClass, domain, range, initialValue) ≡
  let f = new(FUNCTIONELEMENT) in
    fClass(f) := functionClass
  let s = new(SIGNATURE) in
    sigDomain(s) := domain
    sigRange(s) := range
    signature(f) := s
  if initialValue ≠ undef then
    setFunctionValue(f, domain, initialValue)
  add (name, f) to pluginFunctions(SignaturePlugin)

```

The `setFunctionValue` rule sets the value of a function. If the function is not nullary and the specified value is a `MAPELEMENT`, each key in the map is viewed as a function location and the content of the location is set to the corresponding map value.

setFunctionValue

```

setFunctionValue(function, domain, value) ≡
  if isMapElement(value) ∧ domain ≠ undef then
    forall loc ∈ fLocations(value) do
      FSetValue(function, locArgs(loc), fGetValue(value, locArgs(loc)))
  else if domain = undef then
    FSetValue(function, ⟨⟩, value)

```

5.2.2 Universes and Enumerations

To declare universes, the Signature plug-in provides the following patterns:

Universe Declaration

```

( universe xname ) → createUniverse(xname, { })
( universe xname = { xe1, ..., xen } ) → createUniverse(xname, { xe1, ..., xen })

```

The second pattern above allows the specification writer to declare a universe along with a set of named initial member elements. Of course, a declared universe can still be extended using standard methods, namely by using the `extend` rule, which imports a new element

to a universe, or by setting the value of the corresponding universe membership predicate to *true* for a given element.

The universe declaration patterns are interpreted by the `createUniverse` rule, which creates a new universe with the specified name. If initial member elements are specified, for each member element a static function that refers to the member is also created.

createUniverse

```

createUniverse(name, members) ≡
  let u = new(UNIVERSEELEMENT) in
    add (name, u) to pluginUniverses(SignaturePlugin)
  forall elementName ∈ members do
    let e = new(ELEMENT) in
      uMember(u, e) := true
    let f = new(FUNCTIONELEMENT) in
      add (elementName, f) to pluginFunctions(SignaturePlugin)
      fClass(f) := static
      FSetValue(f, ⟨⟩, e)

```

To declare enumerated backgrounds, the Signature plug-in provides the following pattern:

Enumeration Declaration

```

⟨enum xname = {xe1, ..., xen}⟩ → createEnumeration(xname, {xe1, ..., xen})

```

The `createEnumeration` rule is similar in spirit to `createUniverse`, as enumerable backgrounds are analogous to static universes. The rule is as follows:

```

createEnumeration(name, members) ≡
  let b = new(ENUMERATIONBACKGROUND) in
    add (name, b) to pluginBackgrounds(SignaturePlugin)
    forall elementName ∈ members do
      let e = new(ELEMENT) in
        bkg(e) := name
        add e to enumMembers(b)
        let f = new(FUNCTIONELEMENT) in
          add (elementName, f) to pluginFunctions(SignaturePlugin)
          fClass(f) := static
          FSetValue(f, ⟨⟩, e)

```

createEnumeration

Background elements that are defined using the Signature Plug-in are ENUMERATIONBACKGROUND elements. ENUMERATIONBACKGROUND extends BACKGROUND by supporting the Enumerable Interface.

- **controlled** *enumMembers* : ENUMERATIONBACKGROUND → ELEMENT-SET
is the set defining the members of the enumeration. This has a default value of {}.
- *enumerateEnumerationBackground*(*e*) ≡ *enumMembers*(*e*)

5.3 Property Plug-in

The Property Plug-in is a small plug-in that allows correctness properties for a model, expressed as LTL formulas, to be included in the header of a CoreASM specification. Presently, specified properties do not have any meaning during ASM simulations (although it may be possible to extend the Property Plug-in to check simple global assertions). Correctness properties are only applicable during model checking, and are translated by our CoreASM to Promela translator. The details of the translation will be discussed in the next chapter (Section 6.5).

```

(| property  $\alpha$ [e])      → skip
(| check property  $\alpha$ [e]) → skip

```

Property Declaration

CoreASM Operator	Description	Expression Class
G	always	Unary Operator
F	eventually	Unary Operator
U	strong until	Binary Operator
X	next	Unary Operator

Table 5.2: LTL Operators Provided by the Property Plug-in

Including the keyword **check** with a property declaration indicates that the property should be checked during model checking. The **skip** rule is specified as the action for these patterns because they have no semantics in the *CoreASM* interpreter. However, the property expressions are still included as part of the abstract syntax tree. The Property Plug-in also adds the LTL operators listed in Table 5.2 to the *CoreASM* grammar to be used in LTL formulas.

One may see the Property Plug-in as indirectly improving the usability of the Spin model checker, since Spin does not allow LTL properties to be included directly in a specification. In Spin, properties are defined by describing the behavior of a property automaton. Moreover, Spin only allows a single property automaton in each model, while the Property Plug-in allows multiple properties to be specified for a single specification.

Now that we have introduced *CoreASM* and extended the *CoreASM* language to support model checking, we can move on to describing the translation of *CoreASM* specifications to input for Spin in the next chapter.

Chapter 6

From CoreASM to Promela

This chapter presents a novel approach to model checking CoreASM specifications by translating a CoreASM model into an equivalent Promela model, which can be verified using the Spin model checker [36]. From a high level perspective, the steps in the translation and verification process are as follows (see Figure 6.1):

1. A CoreASM specification is loaded and parsed by the CoreASM engine, producing an Abstract Syntax Tree.
2. The Abstract Syntax Tree is translated into Promela.
3. Spin is invoked to generate a verifier of the Promela model, producing C code.
4. The C code is compiled, generating a custom verifier of the CoreASM specification.
5. The verifier is run, producing a counter example if the property being checked is violated.

Using the abstract syntax tree as the basis for translation allows for structured translation and the straight-forward application of a recursive translation procedure.

CoreASM specifications have a well defined structure. Function and universe declarations, as well as correctness properties, are declared in the header section, while the body of a specification consists of rule definitions. The different sections of a specification are translated in the following order:

1. Universe declarations

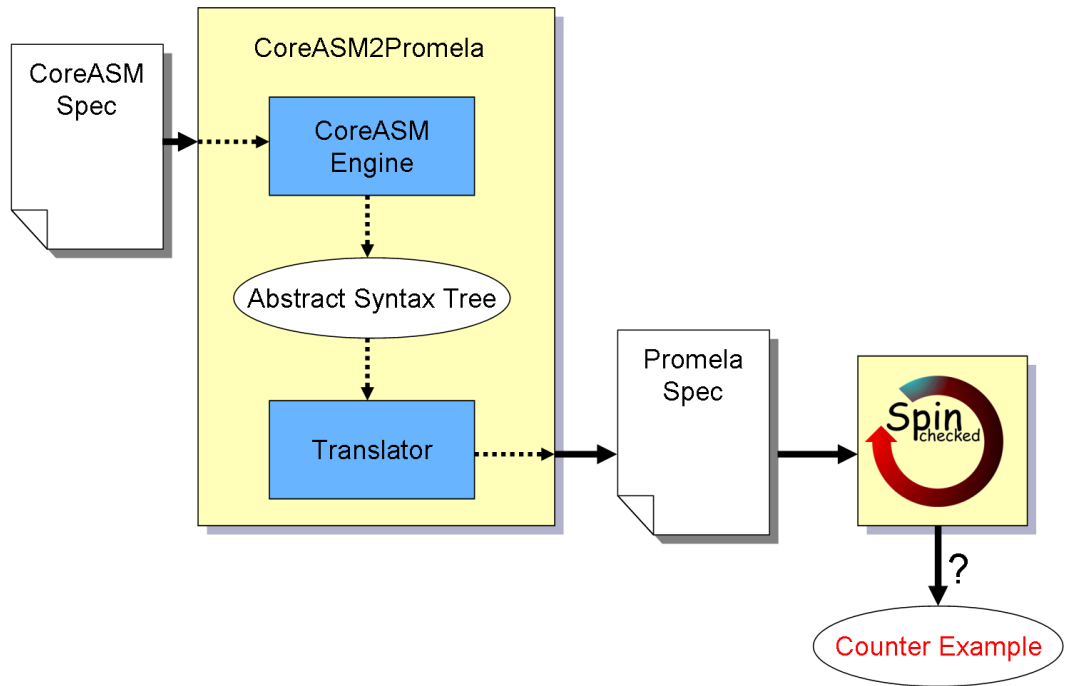


Figure 6.1: CoreASM2Promela: Overall Verification Process

2. Function declarations
3. Correctness properties
4. Rules

Before describing the translation procedure, it will be useful to have a brief introduction to Promela. Promela (Process Meta Language), the input language of the Spin model checker, is a verification modeling language that is based on processes, message channels, and variables. Promela variables have two levels of scope: global and process local. Promela's syntax is similar to C and its execution semantics are also similar to those of C and other imperative programming languages, in that Promela statements are executed sequentially. However, Promela semantics differ from those of regular programming languages in several important regards. Unlike most programming languages, conditions (e.g. $a==b$) are treated as statements in Promela and are only executable when they are true. A false condition blocks execution of the running process. Also, selection and repetition constructs in Promela can be non-deterministic.

The remainder of this chapter presents in detail the CoreASM to Promela translation procedure.

6.1 Elements, Universes and Backgrounds

As Spin can only check finite models, the translation scheme is limited to CoreASM specifications which have finite state as well. Thus the translation only supports only static universes and finite backgrounds, which are handled in exactly the same fashion. Each element from a given universe or background is mapped to an integer value, starting at zero. Since the Boolean elements *true* and *false* are part of the Promela language as well, the CoreASM Boolean background is not translated.

The following CoreASM declarations:

```
universe UniverseName = {universeElement0,...,universeElementN}
enum BackgroundName = {backgroundElement0,...,backgroundElementN}
```

are translated to Promela as:

```
#define universeElement0 0
#define universeElement1 1
...
#define universeElementN N

#define backgroundElement0 0
#define backgroundElement1 1
...
#define backgroundElementN N
```

Under this translation, elements from different universes may have the same underlying numeric value. The translator does not perform any type checking and we assume that all the rules and expressions contained in a specification only refer to elements from equivalent or compatible domains.

The special element *undef* is handled by declaring a macro and we have chosen the convention of declaring the value of *undef* to be the maximum value of the Promela data

type that is used to represent a location. In most situations it suffices to use byte variables to represent locations, making the value of *undef* equal to 255.

```
#define undef 255
```

6.2 Functions

For the translation to be possible all functions referenced in the specification body must first be declared. The translation supports *controlled*, *static*, and *monitored* functions. CoreASM functions are translated as Promela global variables.

Nullary functions are translated as basic integer variables. If f is a nullary function, f is simply translated as a Promela variable:

```
byte f;
```

N-ary functions are translated as multi-dimensional arrays. Each array element corresponds to a unique function location. Although Promela does not directly support multi-dimensional arrays, they can be created indirectly by chaining typedef statements, where each new type contains a single dimensional array, with array size corresponding to the size of the universe of each argument. If f is a function with arity $n \geq 1$ and with arguments (in order) from the universes U_1, \dots, U_n , f is translated as follows:

```
typedef f__ARGn {
    byte ARGn[|U_n|];
}

typedef f__ARG(n-1) {
    f__ARGn ARG(n-1)[|U_(n-1)|];
}

...
```

```

typedef f_ARG1 {
    f_ARG2 ARG1 [| U_1 |];
}

f_ARG1 f;

```

If an initial value for a controlled or static function has been specified in its declaration, the value is set in the initial rule of the ASM, which will be in turn translated to Promela. Rule translation is described in the next section of this chapter.

A controlled function has the default initial value of *undef* for each of its locations. For each controlled function, two variables are declared, one representing the function in the current state and one for the next state. Updates only affect the next-state variable. The suffix '_P' (meaning prime) is added to the variable name to denote that it contains the value of the function in the next state.

```

byte f;
byte f_P;

```

Monitored functions are updated in between each ASM step. A special inline procedure, `monitoredUpdate`, which updates monitored functions, is created. This procedure is called at the beginning of each ASM step. A monitored function's next value is chosen non-deterministically from the values in its range. If g is a monitored function and the values v_1, \dots, v_n are in its range, g is updated by the following conditional statement:

```

inline monitoredUpdate() {
    if
    :: g = v_1;
    :: g = v_2;
    ...
    :: g = v_n;
    fi;

```

```

    ...
}

```

If g has arity greater than zero, each of its locations is updated non-deterministically, in a fashion similar to what is shown above.

6.3 Rules

All basic ASM rules are supported by the translation, except for those which introduce new elements from the reserve, such as **import** and **extend**, since these rules can potentially produce models with infinite state space. The possibility of extending the translation to support those rules will be addressed in the conclusion of this thesis.

6.3.1 Assignment Rule

$loc := value$

The left hand side of the assignment is a location to be updated so in the equivalent Promela statement the primed copy of the variable corresponding to the location is updated.

```
loc_P = value ;
```

6.3.2 Skip Rule

skip

Promela also uses **skip** as its no-op statement.

```
skip ;
```

6.3.3 Block Rule

$\text{par } rule_1 \underbrace{rule_2 \dots rule_n}_{optional} \text{ endpar}$

As in most programming languages, Promela statements are evaluated sequentially. The parallel execution semantics of ASM rules is faithfully modeled since only the primed copy

of variables are modified during rule firing. From here on, the notation $\langle\langle\text{rule}\rangle\rangle$ in the Promela code shall denote the recursive application of the translation algorithm.

```

<<rule1 >>;
<<rule2 >>;
...
<<rulen >>;

```

6.3.4 Conditional Rule

if *value* **then** *rule*₁ **else** *rule*₂
optional

As was mentioned earlier, Promela also has **if** statements as conditional selection constructs. The translation is straight-forward. In the case with no **else** clause, we have:

```

if
:: val -> <<rule1 >>;
:: else -> skip;
fi;

```

otherwise:

```

if
:: val -> <<rule1 >>;
:: else -> <<rule2 >>;
fi;

```

In the first case above, the translation requires the addition an empty **else** clause, as otherwise the execution of the statement in Promela would block when the guard *value* was false. Such behavior would not model the intended ASM execution semantics.

6.3.5 Choose Rule

`choose` x `in` $value$ `with guard` `do` $rule_1$ `ifnone` $rule_2$ `endchoose`
optional *optional* *optional*

A **choose** rule is translated into a non-deterministic Promela conditional statement. Since all universes and backgrounds must be enumerable, it is possible to enumerate all possible values of x in $value$. For each $v_i \in value$, all occurrences of x in $rule_1$ are replaced by v_i . Each of the new rules that results from the substitution becomes a branch of the main conditional statement. In the code below the expression “rule[v/id]” denotes “rule” with all instances of the identifier id replaced with value v .

```
if
:: <<rule1 [v1/id]>>;
:: <<rule1 [v2/id]>>;
...
:: <<rule1 [vn/id]>>;
fi;
```

If the **choose** rule has a **with** clause, the guard, again with the appropriate substitutions of v_i for x , is added as a guard to each of the conditional branches. A final **else** case is added to handle the case when none of the $v_i \in value$ satisfy the guard condition. The Promela statement otherwise would block on such a condition.

```
if
:: guard [v1/id] -> <<rule1 [v1/id]>>;
:: guard [v2/id] -> <<rule1 [v2/id]>>;
...
:: guard [vn/id] -> <<rule1 [vn/id]>>;
:: else -> skip;
fi;
```

Finally, if an **ifnone** clause is provided, the associated rule ($rule_2$) becomes the resultant statement for the **else** case.

```

if
:: guard [v1/id] -> <<rule1 [v1/id]>>;
:: guard [v2/id] -> <<rule1 [v2/id]>>;
...
:: guard [vn/id] -> <<rule1 [vn/id]>>;
:: else -> <<rule2>>;
fi;

```

6.3.6 Forall Rule

forall x in $value$ with $guard$ **do** $rule$ endforall

optional *optional*

Similar to the case for the **choose** rule, a **forall** rule is translated by unrolling it into multiple statements, one for each $v_i \in value$. Again, parallel execution is modeled since only primed variables are updated.

```

<<rule [v1/id]>>;
<<rule [v2/id]>>;
...
<<rule [vn/id]>>;

```

In the case that a **with** clause is included, each statement is translated into a conditional, with the appropriate substitutions made in the *guard*.

```

if
:: guard [v1/id] -> <<rule [v1/id]>>;
:: else -> skip;
fi;
if
:: guard [v2/id] -> <<rule [v2/id]>>;
:: else -> skip;
fi;

```

```

...
if
:: guard[vn/id] -> <<rule[vn/id]>>;
:: else -> skip;
fi;

```

6.3.7 Map Assignment Rule

Since CoreASM functions are objects, we have introduced a rule that assigns a map expression to a function object.

function @f gets $\{(x_{1,1}, \dots, x_{1,m}) \rightarrow y_1, \dots, (x_{n,1}, \dots, x_{n,m}) \rightarrow y_n\}$

This rule is translated into multiple Promela assignment statements, one for each location defined in the map expression.

```

f.ARG1[x_{1,1}]...ARGm[x_{1,m}] = y_1;
...
f.ARG1[x_{n,1}]...ARGm[x_{n,m}] = y_n;

```

6.3.8 Macro Rules

rule ruleName(arg1, ..., argn) = aRule

Macro rules are translated as Promela inline definitions, which model ASM semantics very well, since arguments to inline procedures are passed in a *call-by-substitution* manner.

```

inline ruleName(arg1, ..., argn) {
    <<aRule>>
};

```

6.3.9 Program Rules

```
rule programRuleName = aRule
```

Rules which are agent programs are handled specially. The translation requires that the *program* function be declared as part of the specification signature, and that the function's initial value be specified. If a rule is the value of *program(a)* for some agent *a*, the rule is translated as Promela proctype declaration, which defines a new process type. The special function *self* is defined locally in each process. When a process is instantiated it is given a unique value for *self*, namely the value from the translation of Agent universe declaration. In Promela, statements within an *atomic* block are treated like a single statement. Execution of statements in an atomic block can not be interrupted by another process. This behavior models parallel atomic rule firing in ASMs. In each agent step, monitored functions are updated before the actual program rule is executed. After the program rule is executed, controlled functions are updated, thereby performing the agent's move and producing the next ASM state.

```
proctype programRuleName(byte self) {
    do
        :: atomic {
            monitoredUpdate ();
            <<aRule>>
            functionUpdate ();
        };
    od;
};
```

In the case that we wish to override the purely non-deterministic scheduling policy of Spin, as is the case when we wish to ensure fair scheduling of processes, a mechanism to control the execution of the agent processes is required. Agent execution can be coordinated by using Promela rendezvous channels. A rendezvous channel is a message channel with size zero. A channel wait statement blocks a process' execution until a message is received. We add this channel wait statement to the main loop of an agent to provide a signaling mechanism for the agent. An array of rendezvous channels is defined globally, with one

channel for each agent. A single message type *start* is defined to signal the start of an agent's program execution.

```
mtype = {start};
chan c2p_signal[|Agents|] = [0] of {mtype};
```

```
proctype programRuleName(byte self) {
    do
        :: atomic {
            c2p_signal[self]?start;
            monitoredUpdate();
            <<aRule>>
            functionUpdate();
        };
    od;
};
```

6.4 Expressions

CoreASM2Promela supports the expression types supported in the CoreASM kernel, namely function term expressions and operator expressions. It also supports forall and exists expressions. Function terms are translated into variables or array expressions.

$$f(arg_1, arg_2, \dots, arg_n)$$

```
f.ARG1[ arg_1 ].ARG2[ arg_2 ] . . . . ARGn[ arg_n ]
```

The translation only supports operators which are native to Promela. Table 6.1 lists these operators.

6.4.1 Forall and Exists Expressions

```
forall x in value holds guard
```

CoreASM Operator	Promela Operator	Description
=	==	Equality comparator
!=	!=	Inequality comparator
+	+	Numeric addition
-	-	Numeric subtraction
*	*	Numeric multiplication
/	/	Numeric (integer) division
mod	%	Modulo
<	<	Less than comparator
<=	<=	Less than or equal comparator
>	>	Greater than comparator
>=	>=	Greater than or equal comparator
not	!	Logical negation
and	&&	Logical and
or		Logical or

Table 6.1: CoreASM to Promela Operator Conversion

First-order universally quantified expressions are translated by expanding the expression into a conjunction of predicates over all of the elements in the domain which the expression quantifies over. As is the case with **forall** and **choose** rules, *value* must be a static enumerable domain. If v_1, \dots, v_n are the elements in *value*, a forall expression is translated as:

```
((guard[v1/id]) && ... && (guard[vn/id]))
```

Similarly, existentially quantified expressions are translated as a disjunction of predicates.

exists *x* in *value* with *guard*

```
((guard[v1/id]) || ... || (guard[vn/id]))
```

CoreASM Operator	Promela Operator	Description
not	!	Logical negation
and	&&	Logical and
or		Logical or
implies	- >	Logical implication
G	□	LTL always
F	<>	LTL eventually
U	U	LTL strong until
X	X	LTL next

Table 6.2: Promela LTL Operators

6.5 LTL Correctness Properties

Spin does not support LTL correctness claims directly. LTL properties must be translated into Promela never claims, which correspond to the property automata described in Chapter 3. Only one never claim can appear in a Promela model, so only one property can be checked at a time (though the property may be a conjunction of other properties). Spin provides a tool which converts an LTL formula into an equivalent never claim.

Spin's LTL translator only allows LTL and logical operators to appear in formulas (see Table 6.2), so, for example, comparison expressions cannot be included. However, Spin does support inclusion of complex predicates indirectly through macro definitions. Our translator defines these macros automatically. For example, for the following property:

$$\mathbf{G}(\underbrace{\mathbf{not} \ (owner(resource) = agent1)}_{c2p_prop0c0} \ \mathbf{and} \ \underbrace{owner(resource) = agent2)}_{c2p_prop0c1}) \quad (6.1)$$

the following macros are declared:

```
#define c2p_prop0c0 (owner.ARG1[resource]==agent1)
#define c2p_prop0c1 (owner.ARG1[resource]==agent2)
```

To ensure that a correctness claim is only checked in states where the ASM is properly initialized, a global boolean variable *c2p_initialized* is introduced. Without this variable, the model checker could produce erroneous counterexamples. The value of *c2p_initialized* is initially false and is set to true after the ASM state initialization steps (which are discussed

in the next section) are completed. So to restrict property checking of some property ϕ to initialized states, we check the global implication:

$$\mathbf{G}(c2p_initialized \rightarrow \phi)$$

Moreover, to check that a property ψ holds true in a given model using Spin, we actually need to check that the automaton corresponding to the negation of the property, $\neg\psi$, never accepts. Thus the property that is finally given to the Spin LTL translator is:

$$\neg \mathbf{G}(c2p_initialized \rightarrow \phi)$$

Using the property listed earlier (Equation 6.1) as the example, the property actually given (using Promela LTL syntax) to the Spin LTL translator is:

$$!\!(c2p_initialized \rightarrow (\!\!(c2p_prop0c0 \ \&\& \ c2p_prop0c1))))$$

for which the Spin LTL translator produces the following never claim:

```

never {      /* !!( c2p_initialized ->
                ( !( c2p_prop0c0 && c2p_prop0c1 ) ) ) */
T0_init:
    if
    :: (( c2p_initialized ) && ( c2p_prop0c0 ) && ( c2p_prop0c1 ))
       -> goto accept_all
    :: (( c2p_initialized )) -> goto T0_S4
    :: (1) -> goto T0_init
    fi;
T0_S4:
    if
    :: (( c2p_prop0c0 ) && ( c2p_prop0c1 )) -> goto accept_all
    :: (1) -> goto T0_S4
    fi;
accept_all:
    skip
}

```

6.6 DASM Simulation Model

Every Promela model has a special initial process (which is specified using the keyword *init*) that is the first process run by Spin. In our translation from CoreASM to Promela we use the *init* process to initialize the state of the simulated ASM and, when desired, to explicitly schedule the execution of agents.

In the ASM initialization section of the *init* process, all locations are first given the value *undef*. Then, if the function declaration section of a specification includes initial values, these initial values are set. Afterwards, the actual initial rule given by the specification writer is executed. This rule may perform any other state initialization, which may have not been convenient to express as part of the function declarations. Processes corresponding to the program of each of the agents are then instantiated, with the value of each process' *self* argument set to identify each DASM agent. Note that it is possible for multiple agents to share the same process type. Also, since the agent processes are launched within an atomic sequence, the processes can not begin execution until all the statements in the atomic sequence are complete. Thus all the agent processes begin their execution at the same time. At this point, the ASM is considered to be in its initial state, so the variable *c2p_initialized* is set to true.

```
init {
  atomic {
    functionInit ();
    init_inline ();
    functionUpdate ();
    InitRule ();
    functionUpdate ();
    run program1 (agent1);
    run program2 (agent2);
    ...
    run programN (agentN);
    c2p_initialized = true;
  };
}
```

Spin’s process scheduler is non-deterministic, so during model checking every possible interleaved sequence of process executions is considered. Thus, every possible sequence of DASM agent moves is considered. Interleaving semantics model the partially ordered runs of distributed abstract state machines faithfully, since the coherence condition implies that all linearizations of a partially ordered run result in the same final state.

However, the shortcoming of Spin’s process scheduling model is that it only supports *weak fairness*, not *strong fairness*. Under strong fairness a process that is enabled *infinitely often* will do infinitely many steps, while weak fairness stipulates that a process that is enabled *infinitely long* will do infinitely many steps [36]. Consequently, because of our translation and Spin’s statement execution semantics, it is possible for a DASM agent to never be run. This behavior is not desirable when one wishes to check a model for liveness properties, for which some assumption about fairness is usually made, i.e. all agents will execute infinitely often (no starvation).

We address this problem by explicitly using a fair process scheduling policy when checking a model for liveness. We employ the simplest fair scheduling policy, round-robin. As was mentioned in Section 6.3.9, it is possible to coordinate the execution of Spin processes by signaling via message channels. In this case we add a loop to the Promela init process to explicitly run each agent in sequence.

```

init {
  /* ASM initialization */
  ...

  do:: {
    atomic {
      /* |Agents| is the number of agents */
      c2p_agent = (c2p_agent+1)%(|Agents|);
      c2p_signal[c2p_agent]!start;
    }
  } od;
}

```

While other fair and less restrictive scheduling policies, such as “each agent must be run

at least every k steps”, can be implemented, doing so adds considerable complexity to the Promela model, producing models with much larger state space¹. We leave the option of using custom scheduling policies open to users who are inclined to manually edit a generated Promela model.

Having now presented in detail the method of translating CoreASM specifications into Promela models, which can be verified using the Spin model checker, we move on to presenting applications of this translation in the next chapter.

¹For example, using the “each agent must be run at least every k steps” policy increases the state space by a factor of $k \times |Agents|$.

Chapter 7

Case Studies

In this chapter we present the results of applying our model checking tool on several *CoreASM* specifications¹. We compare our results against those of Winter's *ASM2SMV* model checking tool. We do not compare our tool with Gargantini's *Spin* based tool because the *ASM* language his tool supports is very limited and does not support *DASMs*, and applying the two tools to a simple *ASM* model would produce essentially equivalent *Promela* models. This would not provide an interesting comparison. We first present the specific results from each of the case studies and then give a general discussion and analysis of the results in Section 7.4.

All tests were run on a Sun machine with a 1.2 GHz *UltraSparc* processor and 4 GB of main memory, using *Spin*, version 4.2.8, and *NuSMV*, version 2.4.1.

7.1 Distributed Termination Detection

Eschbach [21] presents a specification and verification of a distributed termination detection algorithm, which was originally proposed by Dijkstra, Feijen, and van Gasteren [19]. Eschbach models the termination detection algorithm as an *ASM* and presents a manual proof of its correctness. In our experiments, we verify the correctness of Eschbach's *ASM* model of the termination detection algorithm using the *CoreASM2Promela* model checking tool.

The problem to be solved is to detect the termination of a computation distributed over

¹Several of these specifications were adapted from *ASM-SL* specifications presented in [53]

several machines (computers), which are connected over a network. We assume that all machines are initially active in the computation. Machines become inactive once they have completed their part of the computation. An active (source) machine may delegate work to another (destination) machine by sending a message to the destination machine. After receiving a message an inactive destination machine becomes active again. A distributed computation is considered to be terminated when all machines are inactive and there are no messages to be processed. Each machine has no knowledge of the activity status and message queue of the other machines. One of the machines in the network is designated as a control machine which monitors the status of the computation. The termination detection algorithm works by sending a token, initiating from the control machine, through the network. Token passing is independent of message passing, and has no impact on the activity status of a machine. When the control machine gets the token back, it can determine if the computation has finished by examining the value of the token. For the details of algorithm we refer the reader to Appendix A.1, which contains the full ASM specification of the algorithm.

The correctness property of most interest for this model is to verify that termination is always detected by the control machine once the computation has terminated.

- P1: If the computation has terminated then termination will eventually be detected.

$$\mathbf{G}(termination \rightarrow \mathbf{F} terminationDetected)$$

Since this is a liveness property, whose correctness is dependent on fair scheduling agents, we use the round robin agent scheduling provided by CoreASM2Promela (see Section 6.6).

As originally presented, Eschbach's ASM specification for termination detection is not amenable to model checking since a machine may send an unbounded number of messages. Thus, our model is modified so that each machine may only send up to a maximum number of messages. In our experiments we varied two parameters, A the number of machines (agents) in the network, and M the maximum number of messages a machine may send. The model checking results are shown in Table 7.1, which lists the time taken to verify that the model satisfies P1. An entry MEM in the table indicates that there was insufficient memory to complete verification, while TR indicates that ASM2SMV translator failed (with segmentation fault).

Parameters	Property	CoreASM2Promela Round Robin	ASM2SMV (BDD)
A=2, M=1	P1	< 1s	< 1s
A=2, M=2	P1	< 1s	1s
A=2, M=3	P1	1s	3s
A=2, M=1024	P1	267s	TR
A=3, M=1	P1	6s	25s
A=3, M=2	P1	61s	269s
A=3, M=3	P1	590s	TR
A=4, M=1	P1	501s	2,884s
A=4, M=2	P1	MEM	TR

Table 7.1: Distributed Termination Detection Model Checking Results

7.2 FLASH Cache Coherence Protocol

The FLASH Cache Coherence Protocol coordinates the sharing of memory among the processing nodes of the Stanford FLASH multiprocessor [42]. Winter used the protocol as a case study in her PhD thesis on ASM2SMV. In the FLASH multiprocessor, distributed memory is partitioned into lines and each line is associated with a home-node which hosts the part of the physical memory where the line resides. The sharing of memory is facilitated by holding local copies of data at each node. The Cache Coherence Protocol guarantees that none of the nodes hold a copy of data that is out of date. The entire specification of the protocol is given in Appendix A.2.

In these experiments, we use an erroneous specification of the protocol, based on the original model Winter used and then corrected, to elicit counterexamples from the model checkers, in addition to verifying a true property. The following properties were tested:

- P2: No two nodes have exclusive access to the same line at any time.

$$\forall i \forall j \neq j' \mathbf{G}(\neg(\text{State}(\text{node}_j, \text{line}_i) = \text{exclusive} \wedge \text{State}(\text{node}_{j'}, \text{line}_i) = \text{exclusive}))$$

- P3: Every request will eventually be acknowledged.

$$\forall i \forall j \mathbf{G}(\text{CurPhase}(\text{node}_j, \text{line}_i) = \text{wait} \rightarrow \mathbf{F}(\text{CurPhase}(\text{node}_j, \text{line}_i) = \text{ready}))$$

- P4: Whenever a node obtains shared access to a line, it will be marked as a sharer of the line.

Parameters	Property	CoreASM2Promela	CoreASM2Promela Round Robin	ASM2SMV (BDD)
N=2, L=1, Q=1	P2 P3 P4	6s N/A 196s	43s 7s 1,894s	438s 921s 76s
N=2, L=2, Q=1	P2 P3 P4	85s N/A 5,187s	5,376s 671s 188,907s	MEM MEM MEM
N=3, L=1, Q=2	P2 P3 P4	164s N/A MEM	16,356s 398s MEM	MEM MEM MEM

Table 7.2: Flash Cache Coherence Protocol Model Checking Results

$$\forall i \forall j \mathbf{G}((\text{State}(\text{node}_j, \text{line}_i) = \text{shared} \rightarrow \mathbf{X}(\text{Sharer}(\text{line}_i, \text{node}_j) = \text{true})) \vee (\text{Sharer}(\text{line}_i, \text{node}_j) = \text{true} \rightarrow \mathbf{X}(\text{State}(\text{node}_j, \text{line}_i) = \text{shared})))$$

P2 and P3 are not satisfied by the model, while P4 holds true. In these experiments, we varied three parameters: N the number of nodes, L the number of lines, and Q the size of the message queue at each node. Also, since P2 and P4 do not require scheduling fairness, we performed the tests both with and without CoreASM2Promela’s explicit round robin scheduling for the sake of comparison. The results of the experiments are shown in Table 7.2.

7.3 i-Protocol

The i-Protocol is an optimized sliding window protocol used in GNU Unix to Unix CoPy (UUCP). Version 1.04 of the i-Protocol contained a non-trivial error and was used as the basis of a comparative study of different verification tools in [20]. Due to the incorrect handling of negative acknowledgements, version 1.04 of the i-Protocol contains a so-called “livelock” error, in which a sequence of packet drops results in a loop where the receiver ignores all subsequent packets from the sender. The full specification of the protocol is given in Appendix A.3.

To check for the presence of the live lock error, we check the model against the following LTL property:

Parameters	Property	CoreASM2Promela Round Robin	ASM2SMV (BDD)
W=1	P4	< 1s	MEM
W=2	P4	1s	TR
W=4	P4	2s	TR

Table 7.3: iProtocol Model Checking Results

- If the packet with sequence number $\lceil W/2 \rceil$ has been accepted and if eventually in the future there are no more data errors or packet drops, then next packet will eventually be accepted.

$$\mathbf{G}(\text{accepted}(\lceil W/2 \rceil) \wedge \mathbf{GF}(\neg dckerr \wedge \neg hckerr \wedge \neg dropped)) \rightarrow \mathbf{F}(\text{accepted}(\lceil W/2 \rceil + 1 \text{ mod } 2W))$$

In our experiments we varied the parameter W , the size of the sending/receiving window. The results are presented in Table 7.3.

7.4 Discussion

The results from the Distributed Termination Detection experiments in Table 7.1 consistently show that verification using CoreASM2Promela and Spin is faster than using ASM2SMV and NuSMV. However, the difference in execution time may be due to the strict round robin scheduling imposed by CoreASM2Promela, which greatly reduces the state space to be search, as compared to the SMV model, which considers all fair runs of the system.

The results from the Flash Cache Coherence Protocol experiments (Table 7.2) show that Spin is faster at finding counterexamples (e.g. P2 and P3 ($N=2$, $L=1$, $Q=1$)) than NuSMV. On the other hand, in this particular test NuSMV was faster than Spin at verifying the true property (i.e. P4 ($N=2$, $L=1$, $Q=1$)). These results confirm the previous results showing that depth-first search algorithms are particularly efficient at finding counterexamples [30]. There is also a notable difference in the performance of Promela models with non-deterministic scheduling and Promela models using round robin scheduling. The better performance of the model with non-deterministic scheduling may partly be explained by the Spin's use of partial order reductions, which can greatly reduce the state space that

needs to be searched during verification. Deterministic round-robin scheduling precludes the application of Spin’s partial order reduction algorithm.

The results from the Flash Cache Coherence Protocol and i-Protocol experiments show several cases where a model produced by CoreASM2Promela can be verified by Spin, while NuSMV runs out of memory when attempting to check the corresponding model produced by ASM2SMV (e.g. Table 7.2, $N=2$, $L=2$, $Q=1$). This behavior may in part be attributed to the algorithms used by the two model checkers. Spin uses an explicit state approach where the reachable states of a system are actually enumerated (by “running” the model). On the other hand, the symbolic model checking approach used by NuSMV manipulates BDD representations of functions which make up the state, and reachable states are determined by placing extra constraints on the BDD formulas. The number of reachable states of a system is often far less than the number of possible states, and since BDDs become increasingly large as more constraints are placed on the functions, the memory usage of symbolic model checking can be quite high. Also, the greater amount of memory used when checking the SMV models may be attributed to inefficiencies in the ASM2SMV translator, which will be discussed ensuingly.

It is unfortunate that more comparisons between our tool and ASM2SMV could not be made due to the failure of the ASM2SMV translator. ASM2SMV’s translation to SMV is not optimal. In the ASM2SMV translation, all function locations are unfolded and represented by individual state variables. ASM rules become guards on updates to these state variables. Overall, this creates very large SMV models when compared to the original ASM specification. Since our CoreASM2Promela translator only unfolds locations when translating forall and choose rules, the Promela models produced are comparable in size to the original ASM specifications.

In summary, the conclusions that can be drawn from our experiments are:

- CoreASM2Promela/Spin finds counterexamples more quickly than ASM2SMV/NuSMV.
- CoreASM2Promela/Spin uses less memory than ASM2SMV/NuSMV, and thus it is possible to verify more (larger) models using CoreASM2Promela/Spin.
- CoreASM2Promela is able to translate larger models than ASM2SMV.

Chapter 8

Conclusion and Future Work

In this thesis we have presented a novel approach to model checking distributed abstract state machines. Model checking support for CoreASM provides a useful tool for ensuring the correctness and improving the quality of ASM software specifications. Our specific accomplishments are as follows:

- We have extended the CoreASM language to include function signatures, thereby effectively adding type information to CoreASM, thus allowing for more concrete specifications. We have also extended CoreASM so that correctness properties can be included as part of a specification.
- We have presented a novel approach for model checking ASMs, by translating CoreASM specifications into Promela models, which can then be verified using the Spin model checker. Our translation supports a much more powerful modeling language than that of the previous work, and most significantly, our translation supports *distributed* abstract state machines.
- We have illustrated the effectiveness of CoreASM2Promela by testing properties of non-trivial models. Our experimental results show that, compared to previous work, our approach is more memory efficient and thus we are able to verify larger models. The results also show that our tool is faster at finding counterexamples for false properties.

We have created a tool that is simple to use, which leverages the power of an existing and widely adopted model checking tool. Our work advances the CoreASM project towards

its goal of providing an open and extensible tool environment for the design, validation, and *verification* of abstract state machines.

8.1 Future Work

There were many ideas and issues encountered during the development of the CoreASM Signature Plug-in and CoreASM2Promela which we did not have the time to fully explore. It would be worthwhile to address these ideas in the future.

The Signature Plug-in presented in Section 5.2 could easily be extended to implement runtime type checking of update sets. For each update to a user declared function, the Signature plug-in would check to see that the update value matches the specified range type of that function. This check would occur during the engine's transition from the *Aggregation* state to the *Step Succeeded* state (see Figure 5.5). Moreover, the Signature Plug-in could be extended with a richer syntax to allow function domains and ranges to be expressed in terms of set operations (union, intersection, difference, etc). Also, supporting map comprehension expressions would provide concise and powerful means of expressing the initial values of functions.

The CoreASM2Promela translation could be extended to support the dynamic introduction of new elements through **import** and **extend** rules, though a constraint would have to be placed on the size of the reserve. Each element in the superuniverse would be associated with a unique integer. Dynamic universes could then be defined in terms of characteristic functions. Moreover, it is possible to support dynamic sets, by defining the contents of a set in terms of a binary containment predicate $contains(x, y)$, where x is a set and y is some element. The practicality of supporting these constructs would need to be investigated, since they would add considerable overhead to a translated Promela model.

The CoreASM2Promela translation algorithm could be optimized. Currently, all locations are refreshed in between each ASM step, regardless of whether they have actually been updated by an ASM rule. This is a relatively expensive operation, which could be optimized by refreshing only those locations which may be updated by the active agent's program. Also, to make the CoreASM2Promela translator more extensible, the translator program could be refactored to adopt a plug-in based architecture, similar to that of the CoreASM engine. In such an architecture, the translation of AST nodes would be delegated to plug-ins, allowing the translator to accept input with an unfixed grammar. Thus, the

translator could easily be extended to handle new rule and expression forms.

One important aspect of model checking ASMs that has not been addressed in this work is the interpretation of counterexamples. This issue is only addressed briefly in Winter's work. Counter-examples produced by a model checking tool are specific execution traces which violate the property being checked. The format of a counterexample is specific to the model checker which produced it. For example, SMV counterexamples begin with an initial state and then list state transitions in terms of updates to state variables, while Spin counterexamples are a complete history of the statements executed in the trace, along with the values of variables. Interpreting counterexamples as ASM runs is not a trivial task. Developing a tool to automate the mapping of model checker counter-examples to ASM runs would be a worthwhile endeavor.

Appendix A

CoreASM Specifications from Case Studies

A.1 Distributed Termination Detection

CoreASM TerminationDetection

```
use StandardPlugins
use MapSetPlugin
use PropertyPlugin
```

```
enum MODE = {SM,RM,P}
enum COLOR = {black, white}
universe Agents = {m0,m1,m2,m3}
```

```
function controlled messages : Agents -> NUMBER
function controlled mode: Agents -> MODE // should actually be monitored
function controlled isActive : Agents -> BOOLEAN
function controlled terminationDetected :-> BOOLEAN initially false
function controlled count : Agents -> NUMBER
function controlled color : Agents -> COLOR
function controlled token : Agents -> BOOLEAN
function controlled tokenColor :-> COLOR initially white
function controlled tokenValue :-> NUMBER initially 0
function static nextMachine : Agents -> Agents
```

```

initially {m0->m3,m1->m0,m2->m1,m3->m2}
function static id : Agents -> NUMBER
initially {m0->0,m1->1,m2->2,m3->3}
function controlled initiateProbe :-> BOOLEAN
initially true // deviating from paper

function static maxMessages :-> NUMBER initially 1
function controlled termination :-> BOOLEAN initially false

function controlled program : Agents -> RULE
initially {m0 -> @Main,m2 -> @Main,m1 -> @Main,m3 -> @Main}

property G(not termination)
property G(terminationDetected implies (G terminationDetected)) //C4
check property G(termination implies (F terminationDetected)) //C3
property G(terminationDetected implies termination) //C2

init InitRule

rule UpdateEnvironment =
par
if isActive(self) then
choose b in BOOLEAN do
isActive(self) := b
endpar

rule SendMessage =
if (mode(self) = SM) then
par
if isActive(self) and (count(self) < maxMessages) then
choose send in BOOLEAN do
if send then
par
choose receivingMachine in Agents with receivingMachine!=self do
messages(receivingMachine) := messages(receivingMachine) + 1
count(self) := count(self) + 1

```

```

        endpar
      endif
    endif
    mode(self) := RM
  endpar

rule ReceiveMessage =
  if (mode(self) = RM) then
    par
      if (messages(self) > 0) then
        par
          messages(self) := messages(self) - 1
          isActive(self) := true
          count(self) := count(self)-1
          color(self) := black
        endpar
        mode(self) := P
      endpar
    endpar

rule Probe =
  if (mode(self) = P) then
    par
      TransmitToken
      InitiateProbe
      NextProbe
      mode(self) := SM
    endpar

rule TransmitToken =
  if token(self) and (isActive(self) = false) and (id(self) != 0) then
    par
      token(self) := false
      token(nextMachine(self)) := true
      if color(self) = black then tokenColor:=black
      tokenValue := tokenValue + count(self)
      color(self) := white
    endpar

```

```
rule InitiateProbe =
if (id(self) = 0) and (initiateProbe = true) then
par
token(nextMachine(self)) := true
tokenValue := 0
tokenColor := white
color(self) := white
initiateProbe := false
endpar

rule NextProbe =
if (id(self) = 0) and token(self) then
    par
if ((count(self) + tokenValue) = 0) and (color(self) = white)
    and (tokenColor = white) and (isActive(self) = false)
then
terminationDetected := true
else
par
initiateProbe := true
token(self) := false
endpar
endif
    endpar

rule Main =
par
SendMessage
ReceiveMessage
Probe
UpdateEnvironment
UpdateTermination
endpar

rule InitRule =
par
```

```

forall a in Agents do
par
program(a) := @Main
mode(a):=SM
  isActive(a):=true
  messages(a):=0
  count(a):=0
  color(a):=white
  token(a):=false
                                endpar
endpar

rule UpdateTermination =
termination := ((messages(m0)+messages(m1)+messages(m2)+messages(m3)) = 0)
              and (forall a in Agents holds isActive(a)=false)

```

A.2 FLASH Cache Coherence Protocol

CoreASM flashProtocol

```

use StandardPlugins
use MapSetPlugin
use PropertyPlugin

universe Agents = { a1, a2, e }
function program : Agents -> RULE
initially {a1->@behavior, a2->@behavior,e->@env}

enum TYPE = { noMess, get, getx, inv, wb, rpl, fwdack, swb,
              invack, nack, nackc, fwdget, fwdgetx, put, putx,
              nackc2, putUswb, putxUfwdack }

enum CCTYPE = { ccget, ccgetx, ccrpl, ccwb }
enum LINE = { l1 }
enum PHASE = { ready, wait, invalidPhase }
enum STATE = { exclusive, shared, invalid }

```

```
function static Home : LINE -> Agents
  initially { l1 -> a1 }

function MessInTr : Agents -> TYPE
  initially { a1 -> noMess, a2 -> noMess }

function SenderInTr : Agents -> Agents
  initially { a1 -> a1, a2 -> a1 }

function SourceInTr : Agents -> Agents
  initially { a1 -> a1, a2 -> a1 }

function static LineInTr : Agents -> LINE
  initially { a1 -> l1, a2 -> l1 }

function SenderInTrR : Agents -> Agents
  initially { a1 -> a2, a2 -> a2}

function SourceInTrR : Agents -> Agents
  initially { a1 -> a1, a2 -> a1 }

function MessInTrR : Agents -> TYPE
  initially { a1 -> noMess, a2 -> noMess}

function static LineInTrR : Agents -> LINE
  initially { a1 -> l1, a2 -> l1 }

function InSender : Agents -> Agents
  initially { a1 -> a2, a2 -> a2 }

function InSource : Agents -> Agents
  initially { a1 -> a2, a2 -> a2 }

function InMess : Agents -> TYPE
  initially { a1 -> noMess, a2 -> noMess }
```

```

function static InLine : Agents -> LINE
  initially { a1 -> l1, a2 -> l1 }

function CurPhase : Agents * LINE -> PHASE
  initially { (a1,l1) -> ready, (a2,l1) -> ready }

function CCState : Agents * LINE -> STATE
  initially { (a1,l1) -> invalid, (a2,l1) -> invalid }

function Pending : LINE -> BOOLEAN
  initially { l1 -> false }

function Owner : LINE -> Agents

function Sharer : LINE * Agents -> BOOLEAN
  initially { (l1,a1) -> false, (l1,a2) -> false }

function monitored produceCCType : Agents -> CCTYPE

property G(not ((CCState(a1,l1) = exclusive) and
                (CCState(a2,l1) = exclusive)))
property G((((CurPhase(a1,l1)=wait) implies F(CurPhase(a1,l1)=ready)))) and
  (G((((CurPhase(a2,l1)=wait) implies F(CurPhase(a2,l1)=ready)))))
property G( ((CCState(a1,l1)=shared) implies X(Sharer(l1,a1)=true)) or
  ((Sharer(l1,a1)=true) implies X(CCState(a1,l1)=shared)) ) and
  G( ((CCState(a2,l1)=shared) implies X(Sharer(l1,a2)=true)) or
  ((Sharer(l1,a2)=true) implies X(CCState(a2,l1)=shared)) )

init Skip

rule AppendToTransit(agentU, senderU, messU, sourceU, lineU) =
  if MessInTr(agentU)=noMess then
    par
      SenderInTr(agentU) := senderU
      MessInTr(agentU) := messU
      SourceInTr(agentU) := sourceU
    endpar

```

```

endif

rule AppendRequestToTransit(agentU, senderU, messU, sourceU, lineU) =
  if MessInTrR(agentU)=noMess then
    par
      SenderInTrR(agentU) := senderU
      MessInTrR(agentU) := messU
      SourceInTrR(agentU) := sourceU
    endpar
  endif

rule R1UR2UR3UR4 =
  if MessInTrR(a1) = noMess then
    par
      if (produceCCType(self)=ccget) and (CurPhase(self,l1)=ready) then
        AppendRequestToTransit (Home(l1),self,get,self,l1)
      endif
      if (produceCCType(self)=ccgetx) and (CurPhase(self,l1)=ready) then
        AppendRequestToTransit(Home(l1),self,getx,self,l1)
      endif
      if (produceCCType(self)=ccrpl) and
        (CurPhase(self,l1)=ready) and
        (CCState(self,l1)=shared) then
        AppendRequestToTransit (Home(l1),self,rpl,self,l1)
      endif
      if (produceCCType(self)=ccwb) and
        (CurPhase(self,l1)=ready) and
        (CCState(self,l1)=exclusive) then
        AppendRequestToTransit(Home(l1),self,wb,self,l1)
      endif
    endpar
  endif

rule R5 =
  if (InMess(self)=get) and (Home(InLine(self))=self) then
    if Pending(InLine(self)) then
      if MessInTr(InSource(self))=noMess then

```

```

    par
        AppendToTransit(InSource(self),self,nack,
            InSource(self),InLine(self))
        InMess(self):=noMess
    endpar
endif
else
    if Owner (InLine(self)) != undef then
        if MessInTr(Owner(InLine(self)))=noMess then
            par
                AppendToTransit(Owner(InLine(self)),self,fwdget,
                    InSource(self),InLine(self))
                Pending(InLine(self)) := true
                InMess(self):=noMess
            endpar
        endif
    else
        if MessInTr(InSource(self))=noMess then
            par
                AppendToTransit(InSource(self),self,put,
                    InSource(self),InLine(self))
                InMess(self):=noMess
                Sharer(InLine(self),InSource(self)) := true
            endpar
        endif
    endif
endif
endif

rule R6 =
    if InMess(self) = fwdget then
        if CCState(self,InLine(self)) = exclusive then
            if Home(InLine(self))=InSource(self) then
                if MessInTr(Home(InLine(self))) = noMess then
                    par
                        AppendToTransit(Home(InLine(self)),self,putUswb,
                            InSource(self),InLine(self))
                    endpar
                endif
            endif
        endif
    endif

```

```

        CCState(self, InLine(self)) := shared
        InMess(self) := noMess
    endpar
endif
else
    if (MessInTr(InSource(self)) = noMess) and
        (MessInTr(Home(InLine(self))) = noMess) then
        par
            AppendToTransit(InSource(self), self, put,
                InSource(self), InLine(self))
            AppendToTransit(Home(InLine(self)), self, swb,
                InSource(self), InLine(self))
            CCState(self, InLine(self)) := shared
            InMess(self) := noMess
        endpar
    endif
endif
else
    if Home(InLine(self)) = InSource(self) then
        if MessInTr(Home(InLine(self))) = noMess then
            par
                AppendToTransit(Home(InLine(self)), self, nackc2,
                    InSource(self), InLine(self))
                InMess(self) := noMess
            endpar
        endif
    else
        if (MessInTr(InSource(self)) = noMess) and
            (MessInTr(Home(InLine(self))) = noMess) then
            par
                AppendToTransit(InSource(self), self, nack,
                    InSource(self), InLine(self))
                AppendToTransit(Home(InLine(self)), self, nackc,
                    InSource(self), InLine(self))
                InMess(self) := noMess
            endpar
        endif
    endif
endif

```

```

        endif
    endif
endif

rule R7 =
  if InMess(self) = put then
    par
      if CurPhase(self, InLine(self)) != invalidPhase then
        CCState(self, InLine(self)) := shared
      endif
      CurPhase(self, InLine(self)) := ready
      InMess(self) := noMess
    endpar
  endif

rule R8 =
  if ((InMess(self) = swb) and (Home(InLine(self)) = self)) then
    par
      Sharer(InLine(self), InSource(self)) := true
      if Owner(InLine(self)) != undef then
        Sharer(InLine(self), Owner(InLine(self))) := true
      endif
      Owner(InLine(self)) := undef
      Pending(InLine(self)) := false
      InMess(self) := noMess
    endpar
  endif

rule R7UR8 =
  if InMess(self) = putUswb then
    par
      if CurPhase(self, InLine(self)) != invalidPhase
        then CCState(self, InLine(self)) := shared
      endif
      CurPhase(self, InLine(self)) := ready
      Sharer(InLine(self), InSource(self)) := true
      if Owner(InLine(self)) != undef

```

```

        then Sharer(InLine(self),Owner(InLine(self))) := true
    endif
    Owner(InLine(self)) := undef
    Pending(InLine(self)) := false
    InMess(self) := noMess
endpar
endif

rule R9 =
    if InMess(self) = nack then
    par
        CurPhase(self,InLine(self)) := ready
        InMess(self) := noMess
    endpar
endif

rule R10 =
    if ((InMess(self) = nackc) and (Home(InLine(self)) = self)) then
    par
        Pending(InLine(self)) := false
        InMess(self) := noMess
    endpar
endif

rule R9UR10 =
    if InMess(self) = nackc2 then
    par
        CurPhase(self,InLine(self)) := ready
        Pending(InLine(self)) := false
        InMess(self) := noMess
    endpar
endif

rule R11 =
    if ((InMess(self) = getx) and (Home(InLine(self)) = self)) then
        if Pending(InLine(self)) = true then
            if MessInTr(InSource(self)) = noMess then

```

```

par
  AppendToTransit(InSource(self),self,nack,
                  InSource(self),InLine(self))
  InMess(self):=noMess
endpar
endif
else
  if Owner(InLine(self)) != undef then
    if MessInTr(Owner(InLine(self))) = noMess then
      par
        AppendToTransit(Owner(InLine(self)),self,fwdgetx,
                        InSource(self),InLine(self))
        Pending(InLine(self)) := true
        InMess(self):=noMess
      endpar
    endif
  else
    if (forall agentU in {a1,a2}
        holds (not(Sharer(InLine(self),agentU))) ) then
      if MessInTr(InSource(self)) = noMess then
        par
          AppendToTransit(InSource(self),self,putx,
                          InSource(self),InLine(self))
          Owner(InLine(self)) := InSource(self)
          InMess(self):=noMess
        endpar
      endif
    else
      if (forall agentU in {a1,a2} holds
          (not(Sharer(InLine(self),agentU)) or
           (MessInTr(agentU) = noMess))) then
        par
          forall agentU in {a1,a2} with (Sharer(InLine(self),agentU)) do
            par
              AppendToTransit(agentU,self,inv,InSource(self),InLine(self))
              Pending(InLine(self)):=true
            endpar
          endforall
        endpar
      endif
    end
  end
end

```

```

        InMess(self):=noMess
    endpar
    endif
endif
endif
endif
endif

rule R12 =
  if InMess(self) = fwdgetx then
    if CCState(self,InLine(self)) = exclusive then
      if (Home(InLine(self))=InSource(self)) then
        if MessInTr(Home(InLine(self))) = noMess then
          par
            AppendToTransit(Home(InLine(self)),self,putxUfwdack,
                            InSource(self),InLine(self))
            CCState(self,InLine(self)):=invalid
            InMess(self):=noMess
          endpar
        endif
      else
        if (MessInTr(InSource(self)) = noMess) and
           (MessInTr(Home(InLine(self))) = noMess) then
          par
            AppendToTransit(InSource(self),self,putx,
                            InSource(self),InLine(self))
            AppendToTransit(Home(InLine(self)),self,fwdack,
                            InSource(self),InLine(self))
            CCState(self,InLine(self)):=invalid
            InMess(self):=noMess
          endpar
        endif
      endif
    else
      if (Home(InLine(self))=InSource(self)) then
        if MessInTr(Home(InLine(self))) = noMess then
          par

```

```

        AppendToTransit(InSource(self),self,nackc2,
                        InSource(self),InLine(self))
        InMess(self):=noMess
    endpar
endif
else
    if (MessInTr(InSource(self)) = noMess) and
        (MessInTr(Home(InLine(self))) = noMess) then
        par
            AppendToTransit(InSource(self),self,nack,
                            InSource(self),InLine(self))
            AppendToTransit(Home(InLine(self)),self,nackc,
                            InSource(self),InLine(self))
            InMess(self):=noMess
        endpar
    endif
endif
endif
endif

rule R13 =
    if InMess(self) = putx then
    par
        CCState(self,InLine(self)) := exclusive
        CurPhase(self,InLine(self)) := ready
        InMess(self) := noMess
    endpar
endif

rule R14 =
    if (InMess(self) = fwdack) and (Home(InLine(self)) = self) then
    par
        Owner(InLine(self)) := InSource(self)
        Pending(InLine(self)) := false
        InMess(self) := noMess
    endpar
endif

```

```

rule R13UR14 =
  if InMess(self) = putxUfwdack then
    par
      CCState(self,InLine(self)) := exclusive
      CurPhase(self,InLine(self)) := ready
      Owner(InLine(self)) := InSource(self)
      Pending(InLine(self)) := false
      InMess(self) := noMess
    endpar
  endif

rule R15 =
  if InMess(self) = inv then
    if MessInTr(Home(InLine(self))) = noMess then
      par
        AppendToTransit(Home(InLine(self)),self,invack,
                        InSource(self),InLine(self))
        InMess(self):=noMess
        if CCState(self,InLine(self)) = shared then
          CCState(self,InLine(self)) := invalid
        else
          if CurPhase(self,InLine(self)) = wait then
            CurPhase(self,InLine(self)) := invalidPhase
          endif
        endif
      endpar
    endif
  endif

rule R16 =
  if (InMess(self) = invack) and (Home(InLine(self)) = self) then
    forall agentU in {a1,a2} do
      if InSender(self)=agentU then
        par
          Sharer(InLine(self),agentU) := false
          if ( forall otherUagentU in {a1,a2} holds

```

```

        (otherUagentU = agentU or
         Sharer(InLine(self),otherUagentU)=false) ) then
    if MessInTr(InSource(self)) = noMess then
    par
        AppendToTransit(InSource(self),self,putx,
                        InSource(self),InLine(self))
        Pending(InLine(self)):=false
        InMess(self):=noMess
    endpar
    endif
else
        InMess(self):=noMess
    endif
endpar
endif
endif

rule R17 =
    if (InMess(self) = rpl) and (Home(InLine(self)) = self) then
    par
        if (Sharer(InLine(self),InSender(self)) = true) and
            (Pending(InLine(self)) = false) then
        par
            Sharer(InLine(self),InSender(self)) := false
            CCState(self,InLine(self)) := invalid
        endpar
        endif
        InMess(self) := noMess
    endpar
    endif

rule R18 =
    if (InMess(self) = wb) and (Home(InLine(self)) = self) then
    par
        if Owner(InLine(self)) != undef then
        par
            Owner(InLine(self)) :=undef
        endpar
        endif
    endpar
    endif

```

```
        CCState(self,InLine(self)) := invalid
    endpar
endif
    InMess(self) := noMess
endpar
endif
```

```
rule behavior =
```

```
    par
        R1UR2UR3UR4
        R5
        R6
        R7
        R8
        R7UR8
        R9
        R10
        R9UR10
        R11
        R12
        R13
        R14
        R13UR14
        R15
        R16
        R17
        R18
    endpar
```

```
rule ClearMessInTr(agentU) =
```

```
    par
        MessInTr(agentU) := noMess
    endpar
```

```
rule SendMess(agentU) =
```

```
    par
        InSender(agentU) := SenderInTr(agentU)
    endpar
```

```

    InMess(agentU) := MessInTr(agentU)
    InSource(agentU) := SourceInTr(agentU)
    ClearMessInTr(agentU)
endpar

rule SendR(agentU) =
  par
    InSender(agentU) := SenderInTrR(agentU)
    InMess(agentU) := MessInTrR(agentU)
    InSource(agentU) := SourceInTrR(agentU)
    MessInTrR(agentU) := noMess
  endpar

rule SendRequest(agentU) =
  if (MessInTrR(agentU) = get) and
    (CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) = ready) and
    (CCState(SenderInTrR(agentU),LineInTrR(agentU)) = invalid) then
    par
      SendR(agentU)
      CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) := wait
    endpar
  else
    if (MessInTrR(agentU) = getx) and
      (CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) = ready) then
      par
        SendR(agentU)
        CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) := wait
      endpar
    else
      if (MessInTrR(agentU) = rpl) and
        (CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) = ready) and
        (CCState(SenderInTrR(agentU),LineInTrR(agentU)) = shared) then
        par
          SendR(agentU)
          CCState(SenderInTrR(agentU),LineInTrR(agentU)) := invalid
        endpar
      else

```

```

    if (MessInTrR (agentU) = wb) and
      (CurPhase(SenderInTrR(agentU),LineInTrR(agentU)) = ready) and
      (CCState(SenderInTrR(agentU),LineInTrR(agentU))=exclusive) then
      par
        SendR(agentU)
        CCState(SenderInTrR(agentU),LineInTrR(agentU)) := invalid
      endpar
    endif
  endif
endif
endif

rule env =
  forall a in {a1,a2} do
    if InMess(a)=noMess then
      if MessInTr(a) != noMess then
        SendMess(a)
      else
        if (MessInTrR(a) != noMess) and (InMess(a)=noMess) then
          SendRequest(a)
        endif
      endif
    endif
  endif

rule Skip =
  skip

```

A.3 i-Protocol

CoreASM iProtocol1

```

use StandardPlugins
use MapSetPlugin
use PropertyPlugin

```

```

universe Agents = { send, recv, e }
function program : Agents -> RULE

```

```

initially {send -> @sendProgram, recv->@recvProgram, e->@env}

enum TYPE = { DATA, ACK, NAK, NOPAK }
enum STATE = { IDLE, SENDuDATA, SENDuACK, ACCEPTuPAK, CHECKuMISSING,
              HANDLEuNAK, DCKERROR, TIMEOUT1, TIMEOUT2 }
enum MODE = { behave, sync }

function static wnd :-> NUMBER initially 2
function static hwnd :-> NUMBER initially 1
function static dest : Agents -> Agents
initially { send -> recv, recv -> send }

function status : Agents -> STATE
initially { send -> IDLE, recv -> IDLE }

function sendsequence : Agents -> NUMBER
initially { send -> 1, recv -> 1 }
function recsequence : Agents -> NUMBER
initially { send -> 0, recv -> 0 }
function rack : Agents -> NUMBER
initially { send -> 0, recv -> 0 }
function lack : Agents -> NUMBER
initially { send -> 0, recv -> 0 }

function recbuf : [0 .. 1] -> BOOLEAN
initially { 0 -> false, 1 -> false }
function nakd : [0 .. 1] -> BOOLEAN
initially { 0 -> false, 1 -> false }
function accepted : [0 .. 1] -> BOOLEAN
initially { 0 -> false, 1 -> false }

function paktype : Agents -> TYPE
initially { send -> NOPAK, recv -> NOPAK }
function paksequence : Agents -> NUMBER
initially { send -> 1, recv -> 1 }
function pakack : Agents -> NUMBER
initially { send -> 0, recv -> 0 }

```

```

function paktypeInTr : Agents -> TYPE
initially { send -> NOPAK, recv -> NOPAK }
function paksequenceInTr : Agents -> NUMBER
initially { send -> 1, recv -> 1 }
function pakackInTr : Agents -> NUMBER
initially { send -> 0, recv -> 0 }

function tmp :-> NUMBER initially 0

function monitored dropped :-> BOOLEAN
function monitored hckerr :-> BOOLEAN
function monitored dckerr :-> BOOLEAN

property G(((accepted(1)=true) and F(G(not (dropped or hckerr or dckerr))))
           implies F(accepted(0)=true))

init Skip

rule putPakToTransit(type, sequence, ack) =
  par
  paktypeInTr(dest(self)) := type
  paksequenceInTr(dest(self)) := sequence
  pakackInTr(dest(self)) := ack
  endpar

rule sendData =
  if paktypeInTr(dest(send)) = NOPAK then
  par
  putPakToTransit(DATA, sendsequence(send), recsequence(send))
  sendsequence(send) := (sendsequence(send)+1)%wnd
  lack(send) := recsequence(send)
  status(send) := IDLE
  endpar
  else
  status(send) := SENDuDATA
  endif

```

```

rule sendAck(sequence) =
  if paktypeInTr(dest(recv)) = NOPAK then
    par
      putPakToTransit(ACK,sequence,sequence)
      lack(recv) := sequence
      status(recv) := IDLE
    endpar
  else
    status(recv) := SENDuACK
  endif

rule acceptPak(sequence) =
  par
    accepted(sequence) := true
    recbuf(sequence) := false
    recsequence(recv) := sequence
    if recbuf((sequence+1)%wnd) then
      status(recv) := ACCEPTuPAK
    else
      if ((sequence+wnd-lack(recv))%wnd) >= (hwnd div 2) then
        sendAck(sequence)
      else
        status(recv) := IDLE
      endif
    endif
  endpar

rule handleMissingPak(sequence) =
  if not(nakd(sequence)) and not(recbuf(sequence)) then
    if paktypeInTr(dest(recv)) = NOPAK then
      par
        putPakToTransit(NAK,sequence,recsequence(recv))
        nakd(sequence) := true
        lack(recv) := recsequence(recv)
        if sequence = ((paksequence(recv)+wnd-1)%wnd) then
          par

```

```

    status(recv) := IDLE
    paktype(recv) := NOPAK
    endpar
    else
    par
tmp := (sequence+1)%wnd
    status(recv) := CHECKuMISSING
    endpar
    endif
    endpar
    else
    par
tmp := sequence
    status(recv) := CHECKuMISSING
    endpar
    endif
    else
if sequence = ((paksequence(recv)+wnd-1)%wnd) then
    par
    status(recv) := IDLE
    paktype(recv) := NOPAK
    endpar
    else
    par
tmp := (sequence+1)%wnd
    status(recv) := CHECKuMISSING
    endpar
    endif
    endif

rule handleDckErr =
    if paktypeInTr(dest(recv)) = NOPAK then
    par
        putPakToTransit(NAK,paksequence(recv),recsequence(recv))
        nakd(paksequence(recv)) := true
        lack(recv) := recsequence(recv)
    status(recv) := IDLE

```

```

paktype(recv) := NOPAK
  endpar
  else
status(recv) := DCKERROR
  endif

rule handleData =
  if (((paksequence(recv)+wnd-lack(recv))%wnd) <= hwnd) and
    (paksequence(recv) != lack(recv)) then
    if dckerr then
      if not(accepted(paksequence(recv))) and
        not(recbuf(paksequence(recv))) and
        not(nakd(paksequence(recv))) then
        handleDckErr
      else
        paktype(recv) := NOPAK
        endif
      else
        par
nakd(paksequence(recv)) := false
if paksequence(recv) = ((recsequence(recv)+1)%wnd) then
  par
    acceptPak(paksequence(recv))
    paktype(recv) := NOPAK
  endpar
  else
    if not(accepted(paksequence(recv))) and
      not(recbuf(paksequence(recv))) then
      par
recbuf(paksequence(recv)) := true
handleMissingPak((recsequence(recv)+1)%wnd)
      endpar
    else
      paktype(recv) := NOPAK
      endif
    endif
  endpar

```

```

    endif
  else
    paktype(recv) := NOPAK
  endif

rule handleAck =
  if (pakack(self) != sendsequence(self)) and
    (((pakack(self)+wnd-rack(self))%wnd) <= hwnd) and
    (((sendsequence(self)+wnd-pakack(self))%wnd) <= hwnd) then
    rack(self) := pakack(self)
  endif

rule handleNak(currurack) =
  if self = send then
    if (paksequence(self) != sendsequence(self)) and
      (((paksequence(self)+wnd-currurack)%wnd) <= hwnd) and
      (((sendsequence(self)+wnd-paksequence(self))%wnd) <= hwnd) then
      if paktypeInTr(dest(self)) = NOPAK then
        par
          putPakToTransit(DATA,paksequence(self),recsequence(self))
          lack(self) := recsequence(self)
        endpar
      status(self) := IDLE
      paktype(self) := NOPAK
    else
      status(self) := HANDLEuNAK
    endif
  else
    paktype(self) := NOPAK
  endif
else
  paktype(self) := NOPAK
endif

rule handlePak =
  if not(hckerr) then
    par

```

```

    handleAck
    if paktype(self) = DATA then
handleData
    else
    if paktype(self) = NAK then
        if (pakack(self) != sendsequence(self)) and
            (((pakack(self)+wnd-rack(self))%wnd) <= hwnd) and
            (((sendsequence(self)+wnd-pakack(self))%wnd) <= hwnd) then
            handleNak(pakack(self))
        else
            handleNak(rack(self))
        endif
    else
        paktype(self) := NOPAK
    endif
endif
endpar
else
    paktype(self) := NOPAK
endif

rule handleTimeout =
    if status(self) != TIMEOUT2 then
    if paktypeInTr(dest(self)) = NOPAK then
        par
            if self = recv then
            par
                forall i in [0 .. 1] with (i != ((recsequence(self)+1)%wnd)) do
                    nakd(i) := false
                    nakd((recsequence(self)+1)%wnd) := true
                endpar
            endif
            putPakToTransit(NAK, (recsequence(self)+1)%wnd, recsequence(self))
            lack(self) := recsequence(self)
            if (self = send) and
                (sendsequence(self) != ((rack(self)+1))%wnd) then
                status(self) := TIMEOUT2
            endif
        endpar
    endif
endif

```

```

        else
status(self) := IDLE
        endif
    endpar
    else
status(self) := TIMEOUT1
    endif
    else
if paktypeInTr(dest(send)) = NOPAK then
    par
putPakToTransit(DATA, (rack(send)+1)%wnd, recsequence(send))
status(send) := IDLE
    endpar
    endif
endif

rule sendProgram =
par
    if ((status(self) = IDLE) and
        (paktype(self) != NOPAK) and
        not((sendsequence(self) != rack(self)) and
            ((sendsequence(self)+wnd-rack(self))%wnd) <= hwnd)))
    then
        handlePak
    endif
    if ((status(self) = SENDuDATA) or
        ((status(self) = IDLE) and
            (sendsequence(self) != rack(self)) and
            (((sendsequence(self)+wnd-rack(self))%wnd) <= hwnd)))
    then
sendData
    endif
    if status(self) = HANDLEuNAK then
handleNak(rack(self))
    endif
    if (status(self) = TIMEOUT1) or
        (status(self) = TIMEOUT2) or

```

```

        ((status(self) = IDLE) and (paktype(self) = NOPAK) and
         not((sendsequence(self) != rack(self))
              and (((sendsequence(self)+wnd-rack(self))%wnd) <= hwnd)))
    then
        handleTimeout
    endif
endpar

rule recvProgram =
par
    if (status(self) = IDLE) and (paktype(self) != NOPAK) then
        handlePak
    endif
    if (status(self) = SENDuACK) then
        sendAck(recsequence(recv))
    endif
    if (status(self) = ACCEPTuPAK) then
        acceptPak((recsequence(self)+1)%wnd)
    endif
    if (status(self) = CHECKuMISSING) then
        handleMissingPak(tmp)
    endif
    if status(self) = HANDLEuNAK then
        handleNak(rack(self))
    endif
    if (status(self) = DCKERROR) then
        handleDckErr
    endif
    if (status(self) = TIMEOUT1) or
        ((status(self) = IDLE) and (paktype(self) = NOPAK))
    then
        handleTimeout
    endif
endpar

rule passPakThru =
    forall t in Agents do

```

```

if (paktypeInTr(t) != NOPAK) and (paktype(t) = NOPAK) then
  par
  if not(dropped) then
    par
    paktype(t) := paktypeInTr(t)
    paksequence(t) := paksequenceInTr(t)
    pakack(t) := pakackInTr(t)
    endpar
  endif
  paktypeInTr(t) := NOPAK
  endpar
endif

rule clearAckPak =
  if lack(recv) <= recsequence(recv) then
    forall i in [0 .. 1] with
      (i <= lack(recv)) or (i > recsequence(recv)) do
        accepted(i) := false
      else
        forall i in [0 .. 1] with
          (i <= lack(recv)) and (i > recsequence(recv)) do
            accepted(i) := false
          endif
        endif
  endif

rule env =
  par
  passPakThru
  clearAckPak
  endpar

rule Skip =
  skip

```

Bibliography

- [1] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.
- [2] C. Beierle, E. Börger, I. Durdanovic, U. Glässer, and E. Riccobene. Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.
- [3] Mordechai Ben-Ari, Zohar Manna, and Amir Pnueli. The temporal logic of branching time. In *POPL*, pages 164–176, 1981.
- [4] A. Benczur, U. Glässer, and T. Lukovszki. Formal Description of a Distributed Location Service for Ad Hoc Mobile Networks. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 - Advances in Theory and Practice*, volume 2589, pages 204–217. Springer, 2003.
- [5] Girish Bhat, Rance Cleaveland, and Alex Groce. Efficient model checking via buchi tableau automata. In *Computer Aided Verification*, pages 38–52, 2001.
- [6] A. Blass and Y. Gurevich. Background, Reserve, and Gandy Machines. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of LNCS, pages 1–17. Springer, 2000.
- [7] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2/3):235–284, May 2005.
- [8] E. Börger, U. Glässer, and W. Müller. Formal Definition of an Abstract VHDL’93 Simulator by EA-Machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.
- [9] E. Börger, P. Päppinghaus, and J. Schmid. Report on a Practical Application of ASMs in Software Design. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of LNCS, pages 361–366. Springer-Verlag, 2000.
- [10] E. Börger, E. Riccobene, and J. Schmid. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.
- [11] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

- [12] William Chan, Richard J. Anderson, Paul Beame, Steve Burns, Francesmary Modugno, David Notkin, and Jon D. Reese. Model Checking Large Software Specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.
- [13] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
- [14] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.
- [15] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
- [16] Alistair Cockburn. Selecting a project’s methodology. *IEEE Softw.*, 17(4):64–71, 2000.
- [17] G. Del Castillo. Towards Comprehensive Tool Support for Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Applied Formal Methods — FM-Trends 98*, volume 1641 of *LNCS*, pages 311–325. Springer-Verlag, 1999.
- [18] G. Del Castillo and K. Winter. Model Checking Support for the ASM High-Level Language. In S. Graf and M. Schwartzbach, editors, *Proceedings of the 6th International Conference TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.
- [19] Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Inf. Process. Lett.*, 16(5):217–219, 1983.
- [20] Yifei Dong, Xiaoqun Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, Scott A. Smolka, Oleg Sokolsky, Eugene W. Stark, and David Scott Warren. Fighting livelock in the i-protocol: A comparative study of verification tools. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 74–88, 1999.
- [21] Robert Eschbach. A termination detection algorithm: Specification and verification. In *World Congress on Formal Methods (2)*, pages 1720–1737, 1999.
- [22] R. Farahbod et al. *The CoreASM Project*. <http://www.coreasm.org>.
- [23] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. In *Proc. of the 12th Int’l Workshop on Abstract State Machines*, 2005.
- [24] R. Farahbod, V. Gervasi, and U. Glässer. Design and Specification of the CoreASM Execution Engine. Technical Report SFU-CMPT-TR-2005-02, Simon Fraser University, February 2005.
- [25] R. Farahbod, V. Gervasi, and U. Glässer. Design and Specification of the CoreASM Execution Engine. Technical report, Simon Fraser University, October 2005. Revised version of SFU-CMPT-TR-2005-02, February 2005.
- [26] R. Farahbod, V. Gervasi, and U. Glässer. Design and Specification of the CoreASM Execution Engine, Part 1: the Kernel. Technical Report SFU-CMPT-TR-2006-09, Simon Fraser University, May 2006. Also available from www.coreasm.org.
- [27] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, (77), March/April 2007. (to be published).
- [28] Martin Fowler. The new methodology. *Software Development magazine*, December 2000.

- [29] A. Gargantini and E. Riccobene. ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation. *Journal of Universal Computer Science*, 7(11):1050–1067, 2001.
- [30] Angelo Gargantini and Constance L. Heitmeyer. Using model checking to generate tests from requirements specifications. In *ESEC / SIGSOFT FSE*, pages 146–162, 1999.
- [31] A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003—Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 2003.
- [32] Rob Gerth, Doron Peled, Moshe Y. Vardi, and Pierre Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *Protocol Specification Testing and Verification*, pages 3–18, Warsaw, Poland, 1995. Chapman & Hall.
- [33] U. Glässer, Y. Gurevich, and M. Veanes. Abstract communication model for distributed systems. *IEEE Trans. on Soft. Eng.*, 30(7):458–472, July 2004.
- [34] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
- [35] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.
- [36] G.J. Holzmann. *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, Reading, Massachusetts, 2003.
- [37] Capers Jones. *Programming productivity*. McGraw-Hill, Inc., New York, NY, USA, 1986.
- [38] Martin Kardos. An approach to model checking asml specifications. In *Proceedings of the 12th International Workshop on Abstract State Machines*, pages 289–304, 2005.
- [39] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.
- [40] Dexter Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
- [41] S. A. Kripke. Semantic analysis of modal logic I: Normal modal and propositional calculi. *Zeitschrift für Mathematische Logik und Grundlagen der Mathematik*, 9:67–96, 1963.
- [42] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The stanford flash multiprocessor. In *ISCA '94: Proceedings of the 21ST annual international symposium on Computer architecture*, pages 302–313, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.
- [43] Kenneth Lauchlin McMillan. *Symbolic model checking: an approach to the state explosion problem*. PhD thesis, Pittsburgh, PA, USA, 1992.
- [44] Mashaal A. Memon. Specification language design concepts: Aggregation and extensibility in coreasm. Master's thesis, Simon Fraser University, Burnaby, Canada, April 2006.
- [45] Microsoft FSE Group. *The Abstract State Machine Language*. Last visited June 2003, <http://research.microsoft.com/fse/asml/>.
- [46] Microsoft FSE Group. *Spec Explorer*. Last visited January 2007, <http://research.microsoft.com/specexplorer/>.
- [47] Bashar Nuseibeh. Soapbox: Ariane 5: Who dunnit? *IEEE Software*, 14(3):15–16, 1997.

- [48] A. Pnueli. A temporal logic of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.
- [49] R. Eschbach and U. Glässer and R. Gotzhein and M. von Löwis and A. Prinz. Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7(11):1024–1049, 2001.
- [50] RTI. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, National Institute of Standards and Technology, Gaithersburg, MD, May 2002.
- [51] Joachim Schmid. *Executing ASM Specifications with AsmGofer*. Last visited Sep. 2005, www.tydo.de/AsmGofer/.
- [52] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
- [53] Calvin K. Tang. Model checking abstract state machines with answer set programming. Master's thesis, Simon Fraser University, Burnaby, Canada, April 2006.
- [54] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Symp. on Logic in Computer Science*, pages 332–344, Cambridge, June 1986.
- [55] J. Christopher Westland. The cost of errors in software development: evidence from industry. *J. Syst. Softw.*, 62(1):1–9, 2002.
- [56] K. Winter. *Model Checking Abstract State Machines*. PhD thesis, Technical University of Berlin, Germany, 2001.