Design and Specification of the CoreASM Execution Engine and Plugins

Engine Version 1.3

(to be released)

Roozbeh Farahbod
info@coreasm.org

Draft: Friday 26th November, 2010 { Criticism welcome.

www.coreasm.org

This document is based on:

R. Farahbod, CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems. Ph.D. thesis. Simon Fraser University, Burnaby, Canada. 258 pp., 2009.

The CoreASM project is the result of a team effort with substantial contributions by Dr. Vincenzo Gervasi, Dr. Uwe Glässer, George Ma, and Mashaal Memon.

Acknowledgments

This work would not have been possible without the kind support and encouragement of Dr. Uwe Glässer and Dr. Vincenzo Gervasi. Also, many ideas in this work are the outcome of lengthy discussions with my dear friends and colleagues Mashaal Memon and George Ma.

I would also like to express my gratitude to Dr. Robert Cameron, Dr. Lou Hafer, Dr. Tom Shermer, and Michael Altenhofen for their valuable feedback and inspiring discussions. I would like to specially thank Dr. Egon Börger for his thorough examination of this work and his suggestions, corrections, and remarks on the theoretical and practical aspects of this work.

License and Copyright Notice

Abstract

Model-based systems engineering naturally requires abstract executable specifications to facilitate simulation and testing in early stages of the system design process. Abstraction and formalization provide effective instruments for establishing critical system requirements by precisely modeling the system prior to construction so that one can analyze and reason about specification and design choices and better understand their implications. There are many approaches to formal modeling of software and hardware systems. Abstract State Machines, or ASMs, are well known for their versatility in computational and mathematical modeling of complex distributed systems with an orientation toward practical applications. They offer a good compromise between declarative, functional and operational views towards modeling of systems. The emphasis on *freedom of abstraction* in ASMs leads to intuitive yet accurate descriptions of the dynamic properties of systems. Since ASMs are in principle executable, the resulting models are validatable and possibly falsifiable by experiment. Finally, the well-defined notion of *step-wise refinement* in ASMs bridges the gap between abstract models and their final implementations.

There is a variety of tools and executable languages available for ASMs, each coming with their own strengths and limitations. Building on these experiences, this work puts forward the design and development of an extensible and executable ASM language and tool architecture, called CoreASM, emphasizing *freedom of experimentation* and *design exploration* in the early phases of the software development process. CoreASM aims at preserving the very idea of ASM modeling — the design of accurate abstract models at the level of abstraction determined by the application domain, while encouraging rapid prototyping of such abstract models for testing and design space exploration. In addition, the extensible language and tool architecture of CoreASM facilitates integration of domain specific concepts and special-purpose tools into its language and modeling environment.

CoreASM has been applied in a broad scope of R&D projects, spanning maritime surveillance, situation analysis, and computational criminology. In light of these applications, we argue that the design and implementation of CoreASM accomplishes its goals; it not only preserves the desirable characteristics of abstract mathematical models, such as conciseness, simplicity and intelligibility, but it also adheres to the methodological guidelines and best practices for ASM modeling.

Contents

Chapter 1

Introduction

Computer-based systems are increasingly integrated into our day-to-day life. They either control or provide platforms for our communication networks, transportation facilities, economic markets, health-care systems, and safety and security facilities. With the increasing complexity of these systems, efficient design and development of high quality computational systems that faithfully conform to their requirements are extremely challenging and the costs of design flaws and system failures are high. Proper understanding of the requirements, precisely documenting design decisions, and effectively communicating such decisions with the domain experts as early as possible play important roles in the design of complex systems. These challenges call for adoption of proper engineering methods and tools and have motivated the use of *formal methods* in software engineering.

Abstraction and formalization provide effective instruments for establishing critical system requirements by precisely modeling systems prior to construction so that one can analyze and reason about specification and design choices and better understand their implications [7]. There are many approaches to formal modelling of software and hardware systems. *Abstract State Machines (ASMs)* [20] are well known for their versatility in computational and mathematical modelling of complex distributed systems with an orientation toward practical applications. The ASM framework offers a universal model of computation and serves as an effective instrument for analyzing and reasoning about complex semantic properties of discrete dynamic systems. For almost two decades, abstract state machines have been studied, practiced, and applied in modeling and specification of systems to bridge the gap between formal and pragmatic approaches. Combining common abstraction principles from computational logic, discrete mathematics, and the concept of transition systems, ASMs have become a well-known method and assumed a major role in providing a solid and flexible mathematical framework for specification and modeling of virtually all kinds of discrete dynamic systems.

In addition, machine assistance plays an increasingly important role in making design and development of complex systems feasible. Abstract executable specifications serve as a basis for design exploration and experimental validation through simulation and testing. Model checking tools based on formal verification techniques help with proving critical properties of systems and assuring \correctness" before deployment.

There is a variety of tools and executable languages available for ASMs, each coming with their own strengths and limitations. In this work, we critically look into their interesting features and potential shortcomings with the goal of understanding the requirements of a modeling language and tool environment that would support high-level design and experimental validation of abstract machine models at the early stages of design and development. Building on these experiences, this work puts forward the design and development of an extensible and executable ASM language and tool architecture, called CoreASM, emphasizing *freedom of experimentation* and *design exploration* in the early phases of the software development process. CoreASM aims at preserving the very idea of ASM modeling | the design of accurate abstract models (*ground models* [12]) at the level of abstraction determined by the application domain, while encouraging rapid prototyping of such models for conformance testing, design space exploration, and experimental validation.

1.1 Towards a Comprehensive Framework

In light of such observations, a question naturally comes to mind: *what does it take to develop a comprehensive framework and tool environment for design and modeling of complex distributed systems and what features should such a framework provide?* Building on our experience with a broad scope of applications spanning web services architectures [35], computational criminology [31], maritime surveillance [36] and situation analysis [30], we believe that the following set of requirements should be satisfied by any such framework:

1. *Simple and concise specifications*
   Specifications written in such a framework should be simple and concise to be readable and understandable by both domain experts and system designers and to facilitate reasoning about the design and the communication of design concepts between those groups.
2. *Precise semantic foundation*
   The modeling language of such a framework should come with a precise semantic foundation as a prerequisite for analysis, validation and verification of the models.
3. *Freedom of abstraction*
   Such a framework should support writing of abstract and minimal specifications that express the original idea behind the designs of systems at the same levels of complexity and enable system designers to stress on the essential aspects of their design rather than encoding the insignificant details.
4. *Design exploration through fast prototyping*
   Exploring the problem space for the purpose of writing an *initial specification*

requires a language that emphasizes freedom of experimentation by minimizing the need for encoding in mapping the problem space to a formal model. This can be achieved by

- *reducing the cost of encoding domain concepts to language concepts* by providing a rich set of abstract data structures, various domain-specific concepts, and extensibility mechanisms for the tool environment and its language,
- *avoiding early commitments* and encouraging rapid prototyping by supporting creation of abstract and untyped models that can later be refined into more concrete models.

5. *Refinement of models*
   Support for abstraction should be paired with a well-defined refinement technique that allows the system designer to cross levels of abstraction and link the models at different levels through incremental steps down to the final implementation (or the concrete model).
6. *Executability of specifications*
   Executability of even fairly abstract and incomplete models is important to allow experimental validation of the specifications at the early stages of design and to improve communication with the stake-holders during the requirements elicitation and analysis process.
7. *Support for distributed models (multi-agent systems)*
   It is only natural to expect a framework for design and modeling of distributed systems to explicitly support distributed and multi-agent design. This includes support for the definition of agent programs (or processes), inter-agent interaction mechanisms, and various scheduling policies.
8. *Non-determinism*
   Non-determinism is useful as a means of abstracting away from details of complicated and potentially deterministic algorithms. For example, non-deterministic descriptions can be used in high-level modeling of the behavior of the environment.

Considering these requirements, we argue that the ASM formalism properly matches our needs as the underlying formal framework for such a tool environment:

- Abstract state machine specifications are in fact rigorously-defined pseudo-code programs on abstract data structures [20]. As a result, they support writing of simple and concise specifications with a precise semantic foundation.
- ASM programs and the data structures can be fairly abstract[1] and yet ASM specifications are in principle executable.

[1] In ASMs arbitrary structures can be used to reflect the underlying notion of state [20, P. 22].

- The ASM framework comes with a sound and powerful notion of step-wise refinement that helps the designer to structure the design of a system into appropriate abstraction levels and link those levels down to the concrete model (or code).
- The ASM formalism supports the design of distributed systems by providing two classes of synchronous and asynchronous multi-agent abstract state machines.
- ASM supports non-determinism in two forms: a *choose* construct that conveniently abstracts from the details of scheduling, and the notion of read-only *monitored functions* that are only updated by the environment of the system.

Looking at past experiences with ASM languages and modeling environments and considering the requirements listed above, we reason that a comprehensive ASM framework for design and analysis of distributed systems should:

1. come with a rich ASM language that supports both basic and distributed ASMs with non-determinism (see Chapter 2);
2. offer a formal (preferably operational) specification of its language and simulation engine that ensures
   - precise semantics,
   - preservation of pure ASM semantics, and
   - executability of the language;
3. ensure freedom of experimentation through extensibility of the language and its environment;
4. support interaction with the environment (e.g., external functions);
5. be implemented as an *open framework* under an open source license[2] and using a platform-independent language and architecture so that it can be later modified or improved as needed by its users.

It would also be an advantage if such a framework provides a GUI (Graphical User Interface) for simulation and debugging. The graphical interface can organize the information relevant to state transitions into different views, visually highlight inconsistencies of the model, and give the user the ability to compare and contrast states and updates produced by different steps.

[2] http://www.opensource.org

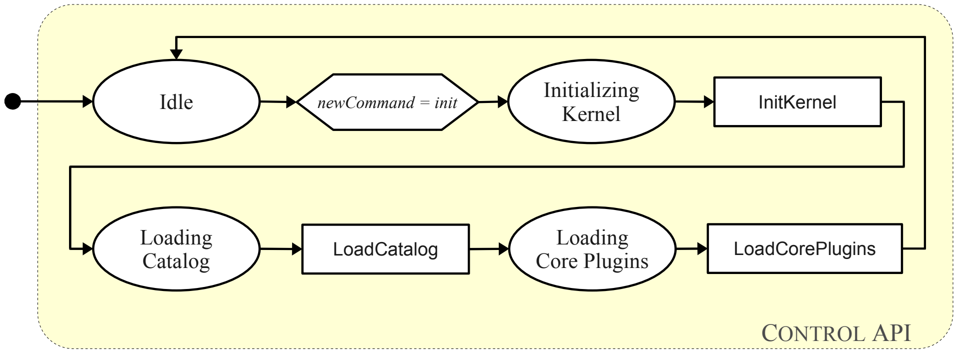


Figure 1.1: An Example of a Control State ASM

1.2 The **CoreASM** Modeling Environment

We take into account the requirements discussed above in the design and development of CoreASM to offer one instantiation of such a comprehensive framework for high-level design and analysis of distributed systems. In this section, we look into different aspects of design and implementation of CoreASM and address some of the challenges one may face during its development.

Formal Specification

There is no need to argue that the development of a reliable modeling framework for design and analysis of distributed systems has to start with a formal (read precise) specification of its language and tool architecture. Abstract state machines have been extensively used for semantic foundations of various programming and system design languages (see Chapter 2). While ASM specifications are primarily operational in nature, they provide a good compromise between declarative, functional and operational views toward modeling of languages and systems. Hence, it is only reasonable to use ASMs in formal modeling of the CoreASM language and its simulation environment (see Chapter 3).

We specify the CoreASM language (both its syntax and the corresponding semantics) through the specification of an interpreter (in form of an abstract state machine), therefore ensuring the executability of the language while providing its formal semantics. The design of the simulation engine and its architecture are specified using Control State ASMs [20], a practical class of abstract state machines that have an easy-to-understand graphical representation (see Figure 1.1 for an example).

Extensible Architecture

In order to provide a rich ASM language that preserves pure ASM semantics and supports sequential and distributed ASMs with non-determinism, we closely follow

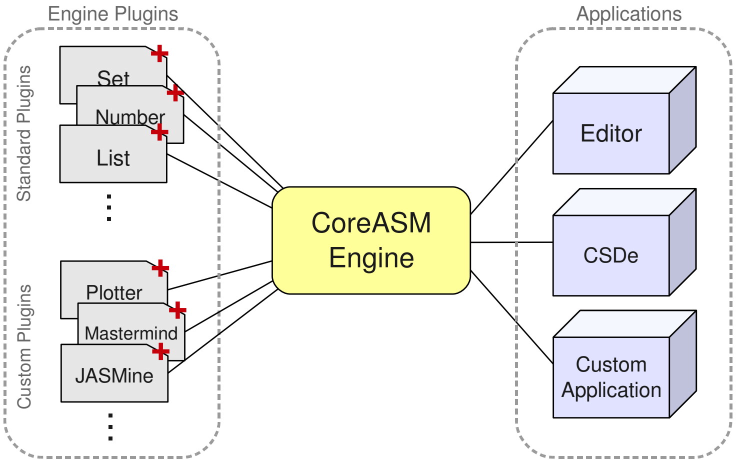


Figure 1.2: CoreASM Extensible Architecture

the formal semantics and the definition of ASMs as provided by the ASM book [20]. However, this may not be enough. ASMs have been used in various domains, some of which required the introduction of special rule forms and data structures into ASMs. To follow the same spirit and to preserve this freedom of experimentation that comes with ASMs, the CoreASM language has to be easily extensible by third parties so that it can naturally fit into different application domains. In addition, to ensure freedom of experimentation, we would like to allow various modeling tools and environments to closely interact with the engine and also to let researchers experiment with variations to the engine's functionality. As a result, we propose a *plugin-based architecture* with a minimal kernel for the CoreASM language and modeling environment to offer the extensibility of both the language and its simulation engine. We start with a micro-kernel (the *core* of the language and its engine) that contains the bare essentials, that is, all that is needed to execute only the most basic ASM. We then implement most of the constructs of the language and the functionalities of the engine through plugins extending the kernel.

Language extensibility is not a new concept [70]. There are a number of programming languages that support some form of extensibility from defining new macros to the definition of new syntactical structures. However, what we are suggesting here is the possibility of extending and modifying the syntax and semantics of the language, keeping only the bare essential parts of the ASM language as static. In order to achieve the this goal, CoreASM plugins should be able to extend the grammar of the core language by providing new grammar rules together with their semantics (see chapters 4 and 5). As a result, every time a CoreASM specification is being loaded, based on the set of plugins that the specification uses, the engine builds a language and a parser for that language to parse the specification. Since the set of all the possible plugins and their grammar rules is not known at the design time (which would otherwise defy the purpose of having a plugin-based architecture) one of the challenges would be to to equip the engine with a fast parser generator capable of generating parsers with look-ahead of more than one to allow the co-existence of more than one grammar rule starting with the same pattern.

Implementation

To facilitate the integration of CoreASM with other complementary tools such as symbolic model checking and automated test generation, the CoreASM engine should have a sophisticated and well defined interface to its environment which provides an API for various operations such as loading a CoreASM specification, starting an ASM run, or performing a single execution step.

In order to have an open and platform-independent implementation of CoreASM, the whole framework is implemented in Java under an open source license (see Chapter 6). After considering various open source license models and looking at similar open source projects, we decided to make CoreASM source code available under the Academic Free License (AFL) version 3.0[3]. AFL 3.0 is an open source license with no reciprocal obligation to disclose source code; i.e., derivative works can be licensed under other licenses, and the source code of those derivative works need not be disclosed. Such a license provides a good compromise between the availability of the original source code in a free form and the existence of potentially proprietary editions and extensions in the industry.

1.3 Related Work

Machine assistance plays an increasingly important role in making practical systems design feasible. Specifically, model-based systems engineering demands for abstract executable specifications as a basis for design exploration and experimental validation through simulation and testing. Thus, it is not surprising that there is a considerable variety of executable ASM languages that have been developed over the years.

The first generation of tools for running ASM models on real machines goes back to Jim Huggins' interpreter written in C [50, 54] and, even further back, to the Prolog-based interpreter by Angelica Kappel [58]. Other interpreters and compilers followed: the lean *EA* compiler [5] from Karlsruhe University, the *scheme*-interpreter [24] from Oslo University, and an experimental EA-to-C++ compiler developed at Paderborn University. Besides practical work on ASM tools, conceptual frameworks for more systematic implementations were developed. The work on the *evolving algebra abstract machine (EAM)* [22], an abstract formal definition of a universal ASM for executing ASM models, contributed to a considerably improved understanding of fundamental aspects of making ASMs executable.

[3] http://www.opensource.org/licenses/afl-3.0.php

Based on such experience, a second generation of more mature ASM tools and tool environments was developed: *AsmL* (ASM Language) [66] and the *Xasm (Extensible ASM) language* [2, 3] are both based on compilers, while the *ASM Workbench* [21], *AsmGofer* [69], and *Asmeta* [39] provide ASM interpreters.

All the above languages build on predefined type concepts rather than the untyped language underlying the theoretical model of ASMs. The most prominent of these languages are Asmeta and AsmL. The Asmeta language, called AsmetaL, implements all the constructs of basic, structured, and multi-agent ASMs as defined in [20], but it is a fully typed ASM language with limited extensibility features. AsmL is a strongly typed language based on the concepts of ASMs but also incorporates numerous object-oriented features and constructs for rapid prototyping of component-oriented software, thus departing in that respect from the theoretical model of ASMs; rather it comes with the richness of a fully fledged programming language. Most of these languages do not provide a run-time system supporting the execution of distributed ASM models[4]; only Xasm (and Asmeta in a limited form) is designed for systematic language extensions; however, the Xasm language itself diverts from the original definition of ASMs and seems closer to a programming language.

For a comprehensive study of related work see [38].

[4] Only Asmeta and AsmGofer provide some sort of support for the execution of distributed ASMs.

Chapter 2

Abstract State Machines

Abstract State Machines (ASMs), originally known as *Evolving Algebras*, were first introduced by Yuri Gurevich [48, 49] as a versatile mathematical method of modeling discrete dynamic systems with the goal of bridging the gap between computation models and specification methods. ASMs combine two well-known and fundamental concepts of *transition systems*, to model the dynamic aspects of a system, and *abstract states*, to model the static aspects at any desired level of abstraction. Egon Börger [20] further developed ASMs into a *systems engineering* method that guides the development of software and embedded hardware-software systems from requirements capture to their implementation.

Today, ASMs are well known for their versatility in computational and mathematical modeling of architectures, languages, protocols and virtually all kinds of sequential, parallel and distributed systems with an orientation towards practical applications. The particular strength of this approach is the flexibility and universality it provides as a mathematical framework for semantic modeling of functional requirements in terms of abstract machine models and their runs. Widely recognized applications of ASMs include semantic foundations of industrial system design languages like the ITU-T standard for SDL [44, 26, 25, 55], the IEEE language VHDL [16, 15] and its successor SystemC [67], programming languages like JAVA [71, 19], C# [14] and Prolog [10, 11], Web service description languages [34, 33, 32], communication architectures[45, 46], embedded control systems [18, 6, 17], et cetera.[1]

In this chapter we briefly recall the basic notions of ASMs as defined in [20] and we use an example to illustrate the application of ASMs with CoreASM in modeling industrial systems.

2.1 Basic ASMs

The original notion of ASMs, or *basic ASMs*, was defined to formalize simultaneous parallel actions of a single computing agent. This notion was later generalized

[1] See also the ASM website at www.asmcenter.org and the overview in [20].

to capture the formalization of multiple agents acting and interacting in an asynchronous manner [20]. In this section, we focus on basic ASMs. *Multi-agent ASMs* or *Distributed ASMs* are explored in the next section.

2.1.1 Basic Definition

A basic ASM M is a tuple of the form (, $\mathcal{I}$, $\mathcal{R}$, P_M) where:

- is a signature; i.e., a finite set of function names f where each function has an *arity*, which is the number of arguments that function takes. Nullary functions, those with arity of zero, are called *constants*. The constants *true*, *false*, and *undef* (representing the \undefined" value) are always defined.
- $\mathcal{I}$ is a set of initial states for signature . A state $\mathfrak{A}$ for is a non-empty set X (the *superuniverse* of $\mathfrak{A}$) together with an interpretation $f^{\mathfrak{A}}$ for each function name f in such that:
  - if f is an n-ary function name, then $f^{\mathfrak{A}} : X^n \mapsto X$, and
  - if c is a constant in , then $c^{\mathfrak{A}} \in X$.

  Functions can be *static* or *dynamic*. Values of dynamic functions can change from state to state.
- $\mathcal{R}$ is a set of rule declarations. In a given state, evaluation of a rule $r \in \mathcal{R}$ produces an *update set* of updates of the form (l, v) where:
  - l is a *location*. A location l in state $\mathfrak{A}$ is a pair $(f, \langle a_1, \ldots, a_n \rangle)$ where f is an n-ary function name in and $a_1, \ldots, a_n$ are values from superuniverse X (i.e., $\forall_{i \in \{1,\ldots,n\}} a_i \in X$). The contents of a location l in $\mathfrak{A}$ is $f^{\mathfrak{A}}(a_1, \ldots, a_n)$.
  - v is a value of superuniverse X.

  The meaning of an update (l, v) is that the content of location l has to be changed to the value v.
- $P_M \in \mathcal{R}$ is a distinguished rule of arity zero (no free variables), called the *main rule* or the *Program* of machine M.

The superuniverse X is usually divided into smaller *universes* modeled by their characteristic functions (unary relations). If D is a universe, then the set of all elements of D is defined as $\{d \mid D(d) = true\}$.

2.1.2 State Transitions

ASM specifications describe how the state of the specified system evolves in time. A computation of M, starting with a given initial state $S_0 \in \mathcal{I}$, results in a finite or infinite sequence of consecutive state transitions of the form

$$S_0 \xrightarrow{\Delta_{S_0}} S_1 \xrightarrow{\Delta_{S_1}} S_2 \xrightarrow{\Delta_{S_2}} \cdots ,$$

such that S_{i+1} is obtained from S_i, for $i \geq 0$, by *firing* Δ_{S_i} on S_i, where Δ_{S_i} denotes a consistent finite set of updates computed by evaluating P_M over S_i.

An update set is called *consistent* if it does not have clashing updates that attempt to assign different values to the same location. The result of firing a consistent update set Δ_{S_i} on S_i is a new state S_{i+1} with the same superuniverse as S_i, such that for every location l of S_i we have:

$$S_{i+1}(l) = \begin{cases} v, & \text{if } (l, v) \in \Delta_{S_i} \\ S_i(l), & \text{otherwise.} \end{cases}$$

2.1.3 Transition Rules

The program P_M of an ASM M is defined by an ASM transition rule.[2] Basic transition rules are as follows:

1. *Skip rule:* **skip**
   Does nothing and evaluates into an empty update set.

2. *Update rule:* $f(a_1, \ldots, a_n) := t$
   Updates the value of $f(a_1, \ldots, a_n)$ to t. It evaluates into an update set of the form $\{(f(a_1, \ldots, a_n), t^{\mathfrak{A}})\}$ where $\mathfrak{A}$ is the current state of the machine and $t^{\mathfrak{A}}$ is the value of t in $\mathfrak{A}$.

3. *Block rule:* P **par** Q
   Evaluates rules P and Q in parallel and the result is the union of the update sets computed by P and Q.

4. *Conditional rule:* **if** ϕ **then** P **else** Q
   If ϕ is true, this rule executes P, otherwise executes Q.

5. *Let rule:* **let** $x = t$ **in** P
   Assigns the value of t to x and executes P. The resulting update set is the update set produced by P.

6. *Forall rule:* **forall** x **with** ϕ **do** P
   Executes P in parallel for every x that satisfies ϕ. The resulting update set is the union of all the update set produced by parallel execution of P over different values of x.

[2] This is a pragmatically generalized definition based on the original definition of an ASM program by [20] which defines an ASM [program] as a set of guarded transition rules.

7. *Choose rule:* **choose** x **with** ϕ **do** P **ifnone** Q
   Non-deterministically (unless otherwise specified) chooses x satisfying ϕ and executes P. If no such x exists, it executes Q.

8. *Sequence rule:* P **seq** Q
   Execute P, if the update set produced by P is consistent, then execute Q in a state which the updates of P are applied. The resulting update set U (based on U_P and U_Q update sets of P and Q) is

$$U = \begin{cases} \{(l,v) \in U_P \mid l \notin \mathit{locations}(U_Q)\} \cup U_Q, & \text{if } U_P \text{ is consistent;} \\ U_P, & \text{otherwise.} \end{cases}$$

9. *Call rule:* $\mathsf{R}(a_1, \ldots, a_n)$
   Execute the previously defined transition rule R with the given parameters. Parameters are passed in a *call-by-name* fashion; i.e., they are passed unevaluated. ASM transition rules can be defined using the expression

$$\mathsf{R}(x_1, \ldots, x_n) = P$$

   where R is the name of the new rule, P is a transition rule and the free variables of P are included in $x_1, \ldots, x_n$.

2.1.4 Interaction with Environment

M interacts with a given operational environment | the part of the external world visible to M | through actions and events as observable at external interfaces, formally represented by externally controlled functions. Intuitively, such functions are manipulated by the external world rather than M itself. Of particular interest are *monitored functions*. Such functions change their values dynamically over runs of M, although they cannot be updated internally by agents of M. A typical example is the abstract representation of global system time. In a given state S of M, the global time (e.g., as measured by some external clock) is given by a monitored nullary function *now*, taking values in a linearly ordered domain $\textsc{Time} \subseteq \textsc{Real}$. Values of *now* increase monotonicly over runs of M.

2.2 Multi-Agent ASMs

Basic ASMs are extended to capture the formalization of multiple agents acting and interacting in an asynchronous manner [20].[3]

An asynchronous multi-agent ASM (or DASM for Distributed ASM) M^D is defined by a dynamic set $\textsc{Agent}$ of computational *agents* each executing its ASM. This

[3] A synchronous version of multi-agent ASMs also exists [20, Sec. 5], in which a set of agents execute their own programs in parallel, synchronized by an implicit global system clock. Since asynchronous ASMs are more general, we will not further explore synchronous ASMs in this survey.

set may change dynamically over runs of M^D, as required to model a varying number of computational resources. Agents of M^D normally interact with one another, and typically also with the operational environment of M^D, by reading and writing shared locations of a global machine state.[4]

A DASM M^D performs a computation step whenever one of its agents performs a computation step. In general, one or more agents may participate in the same computation step of M^D. A single computation step of an individual agent is called a *move*. In this model, moves are atomic. Naturally, conflicting moves must be ordered so that they do not occur in the same step of M^D.

A partially ordered run ρ of M^D is given by a triple $(\;, A, \sigma)$ satisfying the following four conditions (adopted from [49, Sec. 6.5]):[5]

1. is a partially ordered set of moves, where each move has only finitely many predecessors.

2. A is a function on associating agents to moves such that the moves of any single agent of M are linearly ordered.

3. σ assigns a state of M to each initial segment X of , where $\sigma(X)$ is the result of performing all moves in X.

4. *Coherence condition*: If x is a maximal element in a finite initial segment X of 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 $\sigma(Y)$.

A partially ordered run defines a class of admissible runs of M^D rather than a particular run. In general, it may require more than one (even infinitely many) partially ordered run to capture all admissible runs of M^D. From the coherence condition it follows that all *linearizations* of the same finite initial segment of a run of M^D have the same final state.[6] The implication of the partially-ordered-run semantics is illustrated by means of a simple but meaningful example.

Example: Door and Window Manager Assume two propositional variables, *door* and *window*, where *door* = *true* means that `the door is open' and *window* = *true* means that `the window is open'. There are two distinct agents: a door-manager d and a window-manager w.

[4] In principle, one may also compose a DASM of a number of agents, each operating on a part of the state that is disjoint from the view of all the other agents, so that each agent has its own private state.

[5] Here we recall our notes from [29].

[6] Intuitively, a finite initial segment of a partially ordered run ρ is a finite subset of corresponding to a (finite) prefix of ρ.

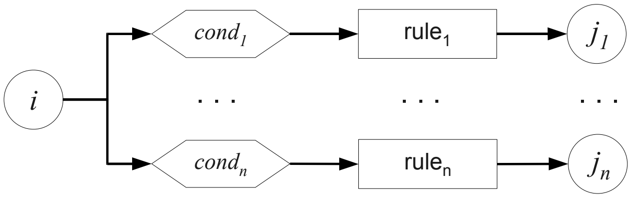


Figure 2.1: Control State ASMs

Door/Window Managers

```
DoorManager ≡
  if $\neg window$ then $door := true$      // move x

WindowManager ≡
  if $\neg door$ then $window := true$      // move y
```

Initially (in state S_0) both the door and the window are closed. Then there are only two possible runs, and in each run only one of the agents makes a move.

We cannot have $x < y$ because w is disabled in the state S_x obtained from S_0 by performing x. Also, we cannot have $y < x$ because d is disabled in the state S_y obtained from S_0 by performing y. Finally, we cannot have a run where x and y are incomparable, that is neither $x < y$ nor $y < x$. By the coherence condition, the final state $S_{x,y}$ of such a run would be obtained from either S_x by performing y or from S_y by performing x; either case is impossible.

2.3 Control State ASMs

In this section we briefly look into *control state ASMs*, a frequently used class of ASMs that represents a normal form of synchronous UML activity diagrams. This particular class of ASMs is expressive enough to model many classical automata such as various extensions of finite state machines, timed automata, push-down automata, etc. It extends finite state machines by synchronous parallelism and by the possibility to also manipulate data [20].

A control state ASM is an ASM whose rules are all of the form presented in Figure 2.1.[7] Such a control state ASM can be formulated in textual form by a parallel composition of Finite State Machine (FSM) rules, where each FSM rule is defined as:

```
FSM($i$, if $cond$ then $rule$, $j$) ≡
  if $ctl\_state = i$ and $cond$ then
    $rule$
    $ctl\_state := j$
```

[7] See [20, Sec. 2.2.6]

Thus, the control state ASM of Figure 2.1 can be formulated as a parallel composition of the following FSM rules:

FSM$(i, \textbf{if } cond_1 \textbf{ then } rule_1, j_1)$
FSM$(i, \textbf{if } cond_2 \textbf{ then } rule_2, j_2)$
...
FSM$(i, \textbf{if } cond_n \textbf{ then } rule_n, j_n)$

Since control state ASMs can be presented in graphical form with a precise semantics, they are a good candidate for documenting functional requirements and modeling of functional aspects of systems at the early stages of design and develop-

The sensors are arranged such that when a train is detected as $coming$, it takes at least d_{min} seconds for it to arrive at the crossing. The gate takes d_{close} seconds to be closed and d_{open} to get opened. Thus, to keep the gate open as much as possible, if we detect a train coming we have $WaitTime = d_{min} - d_{close}$ seconds to start closing the gate. Hence, there is an implicit $deadline$ associated to every track t, indicating the maximum time we have (with regard to track t) in order to safely close the gate.

```
function deadline : Track -> TIME
derived waitTime = dmin - dclose
```

The following nullary function $gateSignal$, controlled by the track control program, signals the opening or closing of the gate.

```
enum GateSignal = {open, close}
function gateSignal : -> GateSignal
```

The Rail Road Crossing ASM consists of two basic ASMs, TrackControl and GateControl, respectively controlling the tracks (sending signals to the gate controller) and maintaining the state of the gate (opening or closing the gate in response to gate signals). We assume that the environment sets the value of the function $trackStatus$ based on the track sensors data.

The track control program TrackControl is a parallel combination of two main rules: 1) closing the gate if needed; i.e., for all tracks, calculating new deadlines, sending a close signal if needed, and clearing passed deadlines; 2) opening the gate if it is safe to do so. The program is defined as follows:[8]

```
rule TrackControl = {
    forall t in Track do {
        SetDeadline(t)
        SignalClose(t)
        ClearDeadline(t)
    }
    SignalOpen
}
```

where we have

```
rule SetDeadline(x) =
    if trackStatus(x) = coming and deadline(x) = infinity then
        deadline(x) := now + waitTime

rule SignalClose(x) =
    if now >= deadline(x) and now <= deadline(x) + 1000 then
        gateSignal := close
```

[8] In CoreASM, curly braces {} can be used to define parallel rule blocks.

```
rule ClearDeadline(x) =
    if trackStatus(x) = empty and deadline(x) < infinity then
        deadline(x) := infinity

rule SignalOpen =
    if gateSignal = close and safeToOpen then
        gateSignal := open
```

The predicate *safeToOpen*, used in the SignalOpen rule, can be defined as follows

$$safeToOpen \equiv \forall t \in \textsc{Track}\ trackStatus = empty \vee deadline(t) > now + d_{open}$$

which is defined in CoreASM as

```
derived safeToOpen = forall t in Track holds
                         trackStatus(t) = empty or deadline(t) > (now + dopen)
```

The gate control program simply responds to gate signals by changing the state of the gate:

```
rule GateControl = {
    if gateSignal = open and gateState = closed then gateState := opened
    if gateSignal = close and gateState = opened then gateState := closed
}
```

2.4.2 The Executable Model

In order to have a meaningful execution of the model, we need to define the initial state of the system and simulate the behavior of the environment. So far we have defined two parallel ASM agents to model track and gate controllers. In this section we add two more agents to our model: an *Environment* agent to model the behavior of the environment and an *Observer* agent to observe the statuses of tracks and the gate and to provide a nicely formatted output throughout the simulation.[9] So, the universe of agents will be defined as:

```
universe Agents = {trackController, gateController, observer, environment}
```

The Environment

The environment agent simulates trains crossing over the tracks in a non-deterministic fashion. If a train is detected as *coming* on a track, we have d_{min} time before it crosses the intersection. Every train takes a certain time to pass the crossing; when that time is reached, the environment sets the track status back to *empty*. The following rule offers one possible definition of such an environment:

[9] However, we do not necessarily need to define these two agents in CoreASM. The environment can be modeled by monitored functions reading input from the user, and the printout can be generated using the Observer plugin presented in Section 5.4.5.

```
rule EnvironmentProgram =
    choose t in Track do {
        if trackStatus(t) = empty then
            if random < 0.05 then {
                trackStatus(t) := coming
                passingTime(t) := now + dmin
            }
        if trackStatus(t) = coming then
            if passingTime(t) < now then {
                trackStatus(t) := crossing
                passingTime(t) := now + 4000
            }
        if trackStatus(t) = crossing then
            if passingTime(t) < now then
                trackStatus(t) := empty
    }
```

The Observer

The observer agent simply prints out the current state of the system. The following observer program prints out the current time, the statuses of all tracks, and finally the state of the gate. To keep the output lines in order, we enclose the print rules in a sequence block.

```
rule ObserverProgram =
    seqblock
        print "Time:  " + (( now - startTime) / 1000) + " seconds"
        forall t in Track do
            print "Track " + t + " is " + trackStatus(t)
        print "Gate is " + gateState
        print ""
    endseqblock
```

The Initial State

In CoreASM, the initial state of the system can be defined in an operational form using an *init* rule. The engine starts the execution of specifications by creating an init agent and assigning the init rule as the program of that agent (see Section 3.2). When the initial state is set up, the init agent can be de-activated by setting its program to *undef* or removing it from the universe of agents.

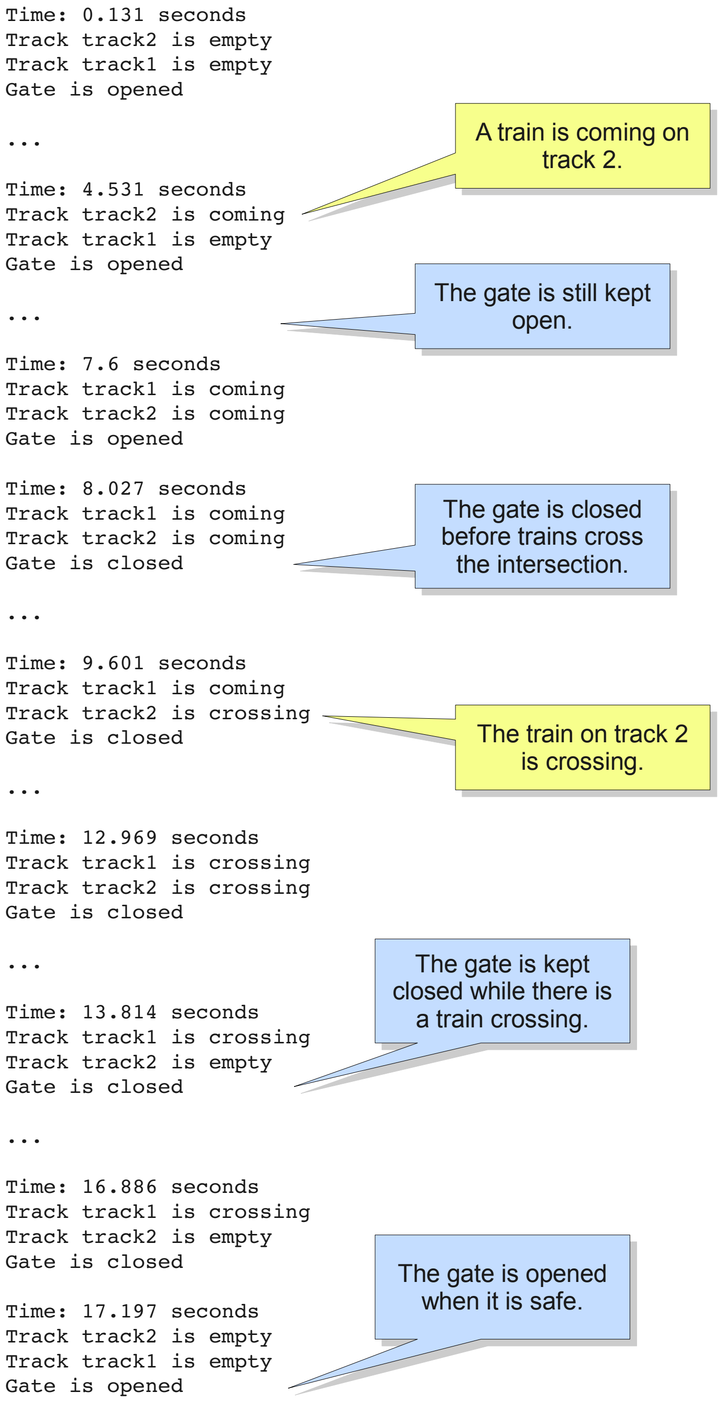
In our example, we assume that initially the gate is open, all the tracks are *empty* and track deadlines are set to positive infinity. The init rule, defined below, sets the initial values of functions and assigns the programs of the agents.

```
init InitRule

rule InitRule = {
    forall t in Track do {
        trackStatus(t) := empty
        deadline(t) := infinity
    }
    gateState:= opened
    dmin:= 5000
    dmax:= 10000
    dopen:= 2000
    dclose:= 2000
    startTime:= now

    program(trackController) := @TrackControl
    program(gateController) := @GateControl
    program(observer) := @ObserverProgram
    program(environment) := @EnvironmentProgram
    program(self) := undef
}
```

The Simulation

Finally, we have everything in place to execute the model in CoreASM and validate the behavior of the gate controller (see Appendix B.1 for the full specification). The execution provides a printout of the states of the system. The output shows that the controller keeps the gate open while there is no train on the tracks and keeps it closed as long as there is at least one train crossing the intersection. Figure 2.2 shows parts of the output of one particular run of the system. As a result of the non-deterministic behavior of the environment, different runs of the model most likely provide different outputs.

It is worth to emphasize that although the ability to execute the model and to observe its behavior enables us to validate the model by experiment, satisfying results of such experiments by no means guarantee the \correctness" of the model. Section 6.3.2 offers a brief discussion on this subject.



```
Time: 0.131 seconds
Track track2 is empty
Track track1 is empty
Gate is opened

...

Time: 4.531 seconds
Track track2 is coming
Track track1 is empty
Gate is opened

...

Time: 7.6 seconds
Track track1 is coming
Track track2 is coming
Gate is opened

Time: 8.027 seconds
Track track1 is coming
Track track2 is coming
Gate is closed

...

Time: 9.601 seconds
Track track1 is coming
Track track2 is crossing
Gate is closed

...

Time: 12.969 seconds
Track track1 is crossing
Track track2 is crossing
Gate is closed

...

Time: 13.814 seconds
Track track1 is crossing
Track track2 is empty
Gate is closed

...

Time: 16.886 seconds
Track track1 is crossing
Track track2 is empty
Gate is closed

Time: 17.197 seconds
Track track2 is empty
Track track1 is empty
Gate is opened
```

A train is coming on track 2.
The gate is still kept open.
The gate is closed before trains cross the intersection.
The train on track 2 is crossing.
The gate is kept closed while there is a train crossing.
The gate is opened when it is safe.

Figure 2.2: Output of the Railroad Crossing Example in CoreASM

Chapter 3

CoreASM: Architectural Overview

The CoreASM language and supporting tool architecture focus on early phases of the software design process. In particular, the goal is to encourage rapid prototyping with ASMs, starting with mathematically-oriented, abstract and untyped models and gradually refining them down to more concrete versions | a powerful technique for specification with refinement that has been exploited in [20] and [13]. In this process, we aim at maintaining executability of even fairly abstract models. Another important characteristic that differentiates our endeavor from previous experiences is the emphasis that we are placing on extensibility of the language. Historical developments have shown how the original, basic definition of ASMs from the Lipari Guide [49] has been extended many times by adding new rule forms (e.g., **choose**) or syntactic sugar (e.g., **case**). At the same time, many significant specifications need to introduce special backgrounds[1], often with non-standard operations. We want to preserve in our language the freedom of experimentation that has proven so fruitful in the development of ASM concepts, and, to this end, we have designed our architecture around the concept of *plugin*s that allows to customize the language to specific needs.

The architecture of the CoreASM engine is partitioned along two dimensions (see Figure 3.1).[2] The first one identifies the main components of the CoreASM engine and their relationships: a *parser*, an *interpreter*, a *scheduler*, and an *abstract storage* (Figure 3.2). We will discuss these components in more detail in Section 3.1. The second dimension, discussed in Section 3.3, distinguishes between what is in the *kernel* of the system | thus implicitly defining the extreme bare bones of the model | and what is instead provided by extension plugins.

[1] We call *background* a collection of related domains and relations packaged together as one logical unit.

[2] This chapter builds on and significantly extends what we have previously published in [28, Section 2].

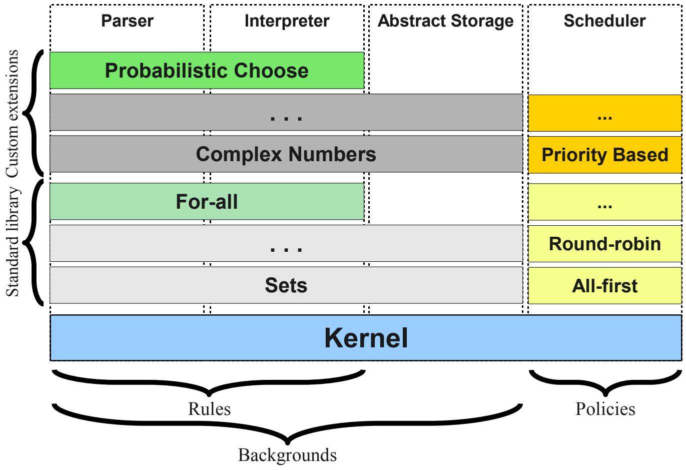


Figure 3.1: Layers and Modules of the CoreASM Engine

These two dimensions correspond to what in the ASM literature have been called *modular decomposition* and *conservative refinement* respectively [13].[3] In particular, our plugins progressively extend (potentially in a conservative way) the capabilities of the language accepted by the CoreASM engine, in the same spirit in which successive layers of the Java [71] and C# [14] languages have been used to structure the language definition into manageable parts.

In this chapter we provide an overview of the architecture of the CoreASM engine and present its components. We also explore the execution lifecycle of the engine and its control state model, and discuss the micro-kernel approach to the design of the engine and its extensibility mechanisms.

3.1 CoreASM Components

The CoreASM engine consists of four components: a parser, an interpreter, a scheduler, and an abstract storage (Figure 3.2). The interpreter, the scheduler, and the abstract storage work together to simulate an ASM run. The engine interacts with the environment through a single interface, called the *Control API*, which provides various operations such as loading a CoreASM specification, starting an ASM run, or performing a single step.

The parser reads a CoreASM specification and generates annotated abstract syntax trees for rules (programs) and definitions of the specification. Each node in these trees may have a reference to the plugin that provides the corresponding syntax. For example, in Figure 3.3, there are nodes that belong to the backgrounds of sets and

[3] While CoreASM plugins are expected to extend the engine mostly through a conservative refinement, the CoreASM architecture does not restrict the plugins to such a refinement.

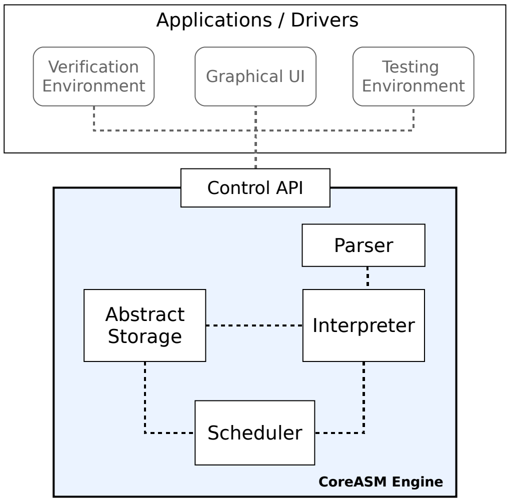


Figure 3.2: Overall Architecture of CoreASM

Booleans; this information will be used by the interpreter and the abstract storage to perform operations on these nodes with respect to the background each node comes from.

The interpreter, executes programs and rules, possibly calling upon background plugins to perform expression evaluation, and upon rules plugins to interpret certain rule forms. It obtains an annotated parse tree from the parser and generates a multiset of *update instructions*, each of which represents either an update, or an arbitrary instruction which will be processed at a later stage by corresponding plugins to generate actual updates (as will be described in more detail on page 39)[4]. The interpreter interacts with the abstract storage to retrieve data from the current state and by executing statements it gradually creates the update set leading to the next state.

The abstract storage manages the data model for the abstract state; in particular, it maintains a representation of the current state of the machine that is being simulated. The state is modeled as a map from locations to opaque elements of a universe Element. The abstract storage also provides interfaces to retrieve values from a given location in the current state and to apply updates. To evaluate a program, the interpreter interacts with the abstract storage in order to obtain values from the current state and generates updates for the next state. In addition, abstract storage also provides auxiliary information about the locations of the current state, such as the ranges and domains of functions or the background to which a particular function or value belongs to.

[4] Where no confusion can arise, in the rest of this document we use the generic term \updates" to refer both to actual updates and to update instructions.

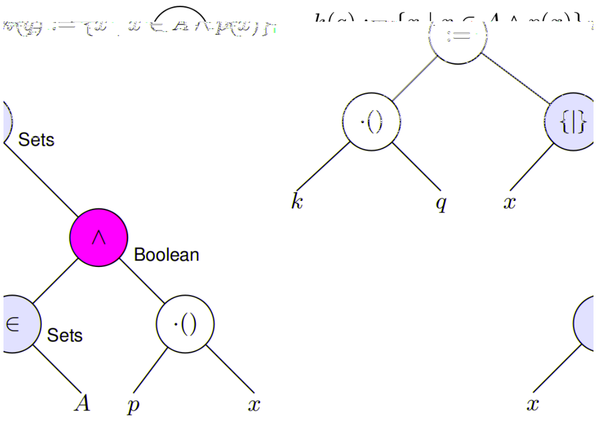


Figure 3.3: Sample Annotated Parse Tree

Finally, the scheduler orchestrates every computation step of an ASM run. In a basic ASM, the scheduler merely arranges the execution of a step: it receives a *step* command from the Control API, invokes the interpreter, and instructs the abstract storage to aggregate the update instructions and *fire* (apply to the state) the resulting update set (if consistent) when the interpreter finishes the evaluation of the program. It then notifies the environment through the Control API of the results of the step.

For distributed ASMs [20], the scheduler also organizes the execution of agents in each computation step. At the beginning of each DASM computation step, the scheduler chooses a subset of agents which will contribute to the next computation step of the machine. The scheduler directly interacts with the abstract storage to retrieve the current set of agents, to assign the current executing agent, and to collect the update set generated by the interpretation of all the agents' programs. Updates are then fired and the environment is notified as for the previous case.

3.2 Engine Lifecycle

The process of executing a CoreASM specification in the CoreASM engine consists of the following steps:

1. Initializing the engine (Figure 3.4)
   (a) Initializing the kernel
   (b) Loading the plugins library catalogue
   (c) Loading and activating core plugins
2. Loading a CoreASM specification (Figure 3.5)
   (a) Parsing the specification header
   (b) Loading required plugins as declared in the specification
   (c) Parsing the specification body

   (d) Initializing the abstract storage
   (e) Setting up the initial state[5]

3. Execution of the specification

   (a) Execute a single step
   (b) If termination condition is not met, repeat from 3a.

The execution process of a single step in the CoreASM engine is as follows (refer also to Figures 3.6 to 3.9 in Section 3.2): The Control API sends a *step* command to the scheduler. *(i)* The scheduler gets the whole set of agents from the abstract storage. *(ii)* It selects a subset of these agents to participate in the next computation step. *(iii)* One by one, the scheduler selects and removes agents from this set and assigns them to the special variable *self* in the abstract storage.[6] *(iv)* The scheduler then calls the interpreter to run the program of the current agent (retrieved by accessing *program(self)* in the current state). *(v)* The interpreter evaluates the program.[7] *(vi)* When the evaluation of the program is complete, the interpreter notifies the scheduler. *(vii)* The scheduler gathers the computed update set and repeats from step (iii) until there is no agent left in the set. When all the agents are executed, the scheduler calls the abstract storage to apply the accumulated updates to the state. *(viii)* If the update set is inconsistent, the abstract storage notifies the scheduler and the notification may lead to selection of a different subset of agents to be executed.[8] If the update set is applied successfully, the Control API is notified of the successful step.

At the end of the execution of each step, the resulting state is optionally made available by the abstract storage module for inspection through the Control API. The termination condition can be set through the user interface of the CoreASM engine, choosing between a number of possibilities (e.g., a given number of steps are executed; no updates are generated; the state does not change after a step; an interrupt signal is sent through the user interface).

In the following sections, we present a high-level but precise specification of the execution process which was presented informally at the beginning of this section. The structure of the specification is that of a control state ASM [20, Sec. 2.2.6][9], as shown in Figures 3.4 to 3.9. The current state of such ASM is given by the variable *engineMode* that controls the execution of rules at any step. The ASM rules corresponding to the control state ASM are also presented.

[5] This ensures that there is at least one agent in the state, the program of that agent being the rule marked with **init** and that agent will contribute to the first step of the simulation.

[6] This is done implicitly by assigning the agent as the value of *executingAgent*. See Section 3.2.3.

[7] This may include a series of interactions between the interpreter and the abstract storage to get values from the current state, which in turn may require interpreting other code fragments, e.g., for derived functions.

[8] The engine can also report (e.g. in a log file) the set of agents whose updates produced an inconsistent update set.

[9] In fact we are using a variant of control state ASMs; see Section 4.5.5 for more details.

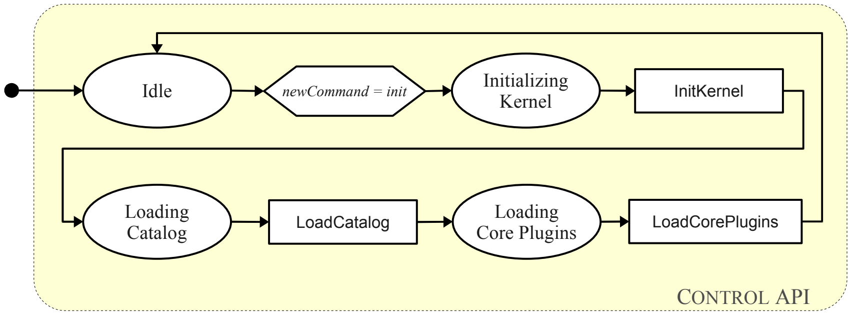


Figure 3.4: Control State ASM of Initializing CoreASM Engine

3.2.1 Engine Initialization

The CoreASM engine starts its execution in the *Idle* state (Figure 3.4). In this state, the engine simply waits for a control command, such as *init* or *step*, from the environment which could be an interactive GUI or a debugger, to start the corresponding task.

Receiving an *init* command (Figure 3.4) will change the state of the engine to *Initializing Kernel* in which the engine initializes its kernel, loads its plugin catalog (the set of all the plugins available to the engine), and finally loads the core plugins. The following rules in Control API abstractly define these tasks. We refer the reader to Section 4.5 for more details on loading plugins.

Control API

```
InitKernel ≡
  $pluginCatalog$ := {}
  $loadedPlugins$ := {}
  $grammarRules$ := {}
  $specification$ := $undef$
  $isStateInitialized$ = $false$

LoadCatalog ≡
  forall $pName$ in $availablePlugins$ do
    let $p$ = $createPlugin(pName)$ in
      add $p$ to $pluginCatalog$

LoadCorePlugins ≡
  forall $p$ in $corePlugins$ do
    LoadPlugin($p$)
```

In order to keep the model consistent, some of the functionalities of the CoreASM kernel can be encapsulated in special *core plugins*. For example, in Section 4.3 we will see how plugins can contribute to the aggregation of updates after every computation step. However, there is also a default aggregation behavior that must be provided

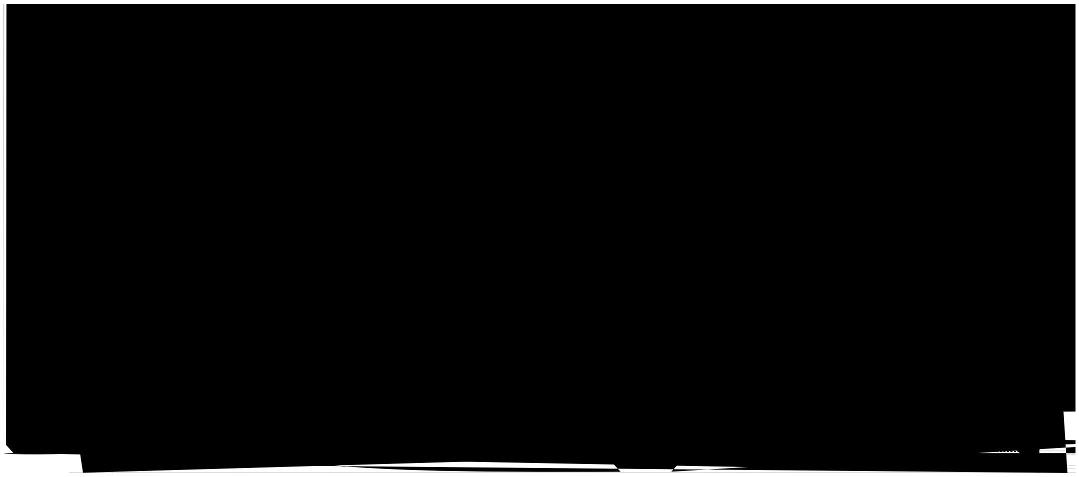


Figure 3.5: Control State ASM of Loading a CoreASM Specification

by the kernel itself. By encapsulating that default behavior in a special core plugin ($Kernel\ plugin$), we are able to reduce the complexity of the aggregation process and specify it in a simple and concise form. So far, the set $corePlugins$ consists of only one plugin; i.e. $corePlugins = \{kernelPlugin\}$.

3.2.2 Loading Specification

Receiving a $load$ command causes the engine to load a new specification (Figure 3.5). The engine first clears previously loaded data, reads the specification file and then parses the specification header to get the list of specific plugins required to be loaded.

Control API

```
**ClearLoadedData** $\equiv$
  **if** $specHasBeenLoaded$ **then**
      **seq**
        $loadedPlugins := \{\}$
        $grammarRules := \{\}$
        $specification := getSpecification(newCommand)$
      **next**
        LoadCorePlugins
where
  $specHasBeenLoaded \equiv |loadedPlugins| > |corePlugins|$
```

Parser

```
**ParseHeader** $\equiv$
  $specPlugins := requestedPlugins(specification)$
```

Loading the required plugins is done in two steps. First, all the package plugins

(plugins that are basically a set of other plugins) are expanded and their enclosed plugins are added to the list of required plugins. In the next step, plugins are loaded one by one according to their loading priority.

Control API

```
LoadSpecPlugins ≡
  seq
  // 1. expanding package plugins
    forall $p$ in $specPlugins$ do
      if $isPackagePlugin(p)$ then
        forall $p'$ in $enclosedPlugins(p)$ do
          add $p'$ to $specPlugins$
  next
  // 2. loading plugins with the maximum load priority first
    while $|specPlugins \setminus loadedPlugins| > 0$ do
      let $toLoad = specPlugins \setminus loadedPlugins$ in
        choose $p$ in $toLoad$ with $maxPriority(p, toLoad)$ do
          if $requiredPlugins(p) \subset specPlugins$ then
            LoadPlugin($p$)
          else
            Error(`Cannot load plugin.')
```

After all the required plugins are loaded, the specification is parsed using the grammar rules provided by the plugins. The root node of the resulting parse tree is kept for future references.

Parser

```
ParseSpecification ≡
  $rootNode(specification) \leftarrow$ Parse($specification, grammarRules$)
```

To prepare the engine for the first simulation step, Abstract Storage is initialized taking into account all plugins contributions, such as backgrounds, universes, functions, and macro rules. A universe of *Agents* and a function *program* that assigns programs to agents are also created in this step. See page 67 for the definition of LoadVocabularyPlugins.

Abstract Storage

```
InitAbstractStorage ≡
  let $newState = new(\textsc{State})$ in
    $state := newState$
    InitializeState($newState$)
    LoadVocabularyPlugins($newState$)
```

```
**InitializeState**($state$) $\equiv$
  **let** $u$ = $new$(UniverseElement) **in**
    $stateUniverse(state,$ "Agents") := $u$
  **let** $f$ = $new$(FunctionElement) **in**
    $stateFunction(state,$ "program") := $f$
  $executingAgent$ := $undef$      // holds the value of `$self$' in the simulated machine
  $stepCount$ := 0
```

Finally, an initial state is created with at least one agent that, in the first step of the simulation, will run the **init** rule as its main program. In addition, based on the set of plugins used by the specification, a scheduling policy will also be chosen by the scheduler.[10]

Scheduler

```
**PrepareInitialState** $\equiv$
  LoadSchedulingPolicy
  **let** $a$ = $new$(Element) **in**
    $initAgent$ := $a$
    SetValue(("Agents", $\langle a \rangle$), true$_e$)
    SetValue(("program", $\langle a \rangle$), $initRule$)
```

Alternatively, an external application may ask the engine to only *parse* the specification (and not loading it). This is useful when an application needs to use only the parsing functionality of the engine, for example to work on a parse-tree view of a specification. In this case, the last two steps of initializing state and preparing the initial state will be skipped. Also, an application can query the list of plugins required by a given specification by sending a *parseHeader* command. In this case, the engine does not parse the specification and stops after loading the required plugins.

3.2.3 Execution of Specification

A *step* command triggers the start of a computation step; this is performed by changing the control state to *Starting Step* which then transfers the control flow to the scheduler.

The StartStep rule in the scheduler initializes *updateInstructions* (the multiset of accumulated update instructions for the current step) and *selectedAgentsSet* (the set of agents selected to perform computation in the current step) and assigns the current set of agents in the simulated machine to *agentSet* by querying the abstract storage module for the current value of *Agents* and only picking those agents whose program is not undefined. We model the query process through the abstract function $getValue(l)$ which takes a location l and retrieves the value of the location from the simulated state. We use the notation "term" to denote the quoted variable or literal term *term* in the simulated machine. Based on the retrieved set of agents, a new

[10] We refer the reader to Appendix A.3 for more details.

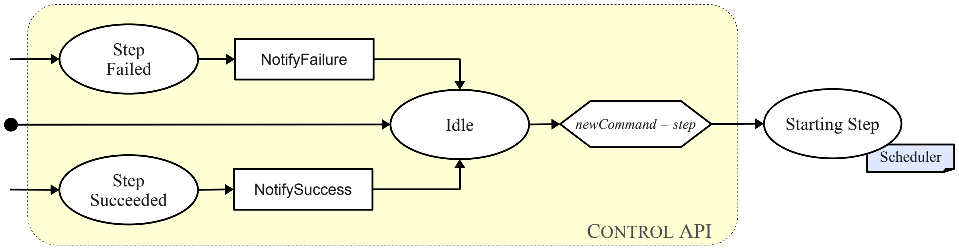


Figure 3.6: Control State ASM of a $step$ command: Control API Module

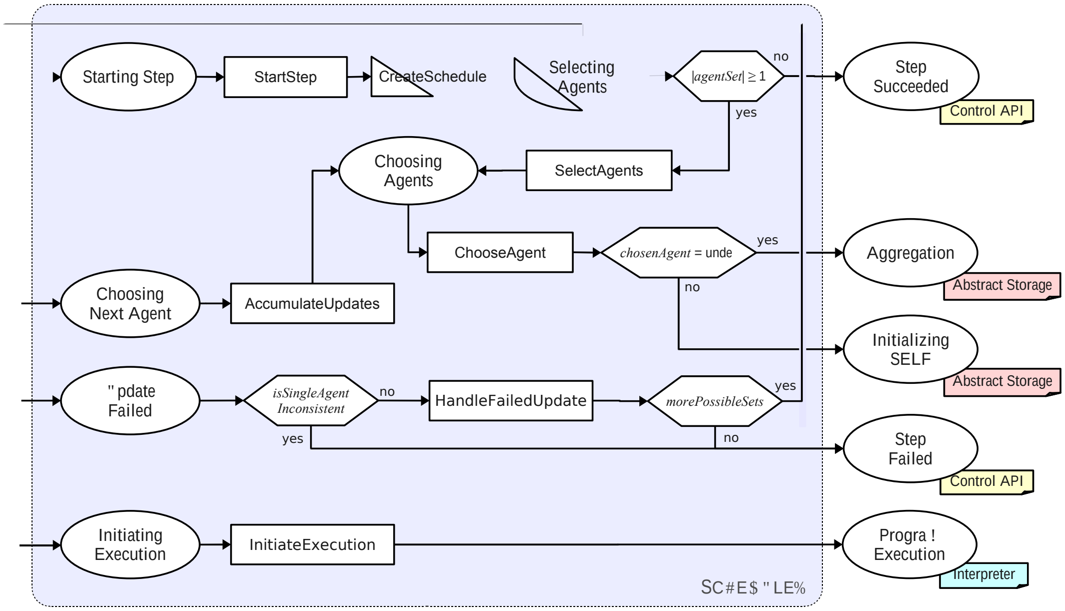


Figure 3.7: Control State ASM of a $step$ command : Scheduler

schedule is then created by CreateSchedule. The control state is then changed to *Selecting Agents*.

Scheduler

```
StartStep ≡
    $updateInstructions := \{\!|\,|\!\}$
    $selectedAgentsSet := \{\}$
    if $stepCount < 1$ then
        $agentSet := \{initAgent\}$
    else
        $agentSet := \{a | a \in getValue((\text{“Agents”}, \langle\rangle)) \wedge getValue((\text{“program”}, \langle a \rangle)) \neq \mathsf{undef}_e\}$
```

```
CreateSchedule ≡
  if $schedulingPolicy \neq undef$ then
    let $R = newScheduleRule(schedulingPolicy)$ in
      $schedule \leftarrow R(schedulingGroup, agentSet)$
```

In the *Selecting Agents* state, if no agent is available to perform computation, the step is considered complete; otherwise, the SelectAgents rule chooses a set of agents to execute in the current step. If there is no scheduling policy provided by any of the plugins, a non-deterministic subset of the agents is chosen; otherwise, the selected agents will be determined by the current scheduling policy. The ChooseAgent rule chooses an agent from this set and changes the state to *Initializing SELF* which leads to the execution of the SetChosenAgent rule in the abstract storage module. After the execution of the agent, the computed updates are accumulated by AccumulateUpdates rule in the *Choosing Next Agent* state, and control state is changed back to *Choosing Agent* until all selected agents have been executed.

Scheduler

```
SelectAgents ≡
  if $schedulingPolicy = undef$ then
    choose $s$ with $s \subseteq agentSet \wedge |s| \geq 1$ do
      $selectedAgentsSet := s$
  else
    $selectedAgentsSet := head(schedule)$
    $schedule := tail(schedule)$

ChooseAgent ≡
  choose $a$ in $selectedAgentsSet$ do
    remove $a$ from $selectedAgentsSet$
    $chosenAgent := a$
  ifnone
    $chosenAgent := undef$

AccumulateUpdates ≡
  add $updates(root(chosenProgram))$ to $updateInstructions$
```

Two rules in the abstract storage module take care of setting the chosen agent and of retrieving the program associated with the chosen agent (by accessing *program*(*self*) in the simulated state). Control then moves back to the scheduler at *Initiating Execution*.

Abstract Storage

```
SetChosenAgent ≡
  $executingAgent := chosenAgent$

GetChosenProgram ≡
  $chosenProgram := getValue((\text{“program”}, \langle executingAgent \rangle))$
```

Figure 3.8: Control State ASM of a *step* command : Abstract Storage

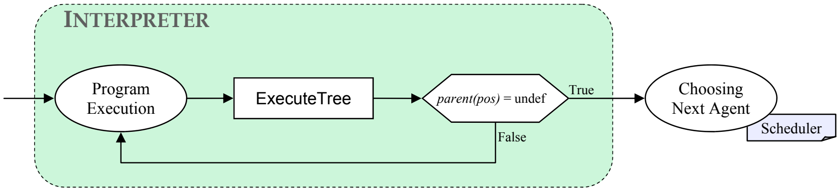


Figure 3.9: Control State ASM of a *step* command : Interpreter

The execution of the program of the chosen agent is initiated in the *Initiating Execution* state in the scheduler and then starts in the *Program Execution* state in the interpreter. During the execution, computed update instructions are progressively added to *updateInstructions*, and when all selected agents have performed their computation, control moves to *Aggregation* state in the abstract storage, where the final update set is calculated and then applied to the current state.

Extending the basic idea presented in [71], we interpret a program by associating values, updates and locations to nodes in the parse tree of the program. Before actually starting the interpreter, previously computed values are removed by the InitiateExecution rule, and the current position in the tree (denoted by the nullary function *pos*) is initialized to the root node of the tree that represents the current program (that is, the program of the current agent, as established above).

Scheduler

```
InitiateExecution ≡
  let $p$ = $root(chosenProgram)$ in
    ClearTree($p$)
    $pos := p$
```

The specification of the interpreter is explored in detail in Section 4.2. We do not include here the full specification for the interpreter; we show instead its most interesting feature, that is the way it interacts with rule and background plugins to delegate interpretation of the associated extensions. To do this, we slightly extend the ASM framework to include ASM rules (programs) as elements of the state; i.e. we assume that ASM rules are elements of the domain Rule and that they can be treated as terms and so can be assigned as values of functions.

As already discussed earlier, nodes of the parse tree corresponding to grammar rules provided by a plugin are annotated with the plugin's identifier. The annotation process is done during parsing, but here we abstract from the details of how it is implemented, and use instead an oracle function $plugin(node)$ for this purpose. While interpreting the parse tree (see ExecuteTree below), if a node is found to refer to a plugin, rules provided by that plugin are obtained through the $pluginRule$ function and executed; otherwise, the kernel interpreter rules (see Section 4.2) are used. Results of the interpretation of node pos are stored alongside the node, and accessed by three functions: $value(pos)$ returns the computed value for an expression node, $updates(pos)$ returns the set of updates generated by a rule node, and $loc(pos)$ returns the location denoted by the node (which is used as lhs-value for assignments). Section 4.2.1 presents a more precise definition of these functions.

Interpreter

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

After executing the programs of all the selected agents, all the update instructions will have been accumulated in $updateInstructions$. Control will move from *Choosing Agent* in the scheduler to *Aggregation* in the abstract storage module. In the *Aggregation* state, the abstract storage aggregates update instructions to compute updates on the locations of the state (see Section 4.3.2 for details), checks the consistency of the computed updates (possibly interacting with the relevant background plugins to evaluate equality), and either applies the updates to the current state through FireUpdateSet (thus obtaining the next state), or provides an indication of failure by changing the state to *Update Failed*.

Abstract Storage

```
AggregateUpdates ≡
  $updateSet \leftarrow$ Aggregate($updateInstructions$)

FireUpdateSet ≡
  forall $(l, v) \in updateSet$ do
    SetValue($l, v$)
```

In the earlier versions of CoreASM [27], if an inconsistent set of updates would be generated in a step, the HandleFailedUpdate rule in the scheduler module would prepare a different subset of agents for execution, and the step would be re-initiated. As a result, if a single agent would produce inconsistent updates, instead of reporting the inconsistency as an error, that agent would be removed from the set of computing agents. We later improved the control flow so that an update fails if the inconsistent set of updates are produced by a single agent. Otherwise, if the inconsistency is between two updates from two different agents, other combinations of agents are tried and the process is iterated until either a consistent set of updates is generated, in which case the computation proceeds to the *Step Succeeded* state of the Control API, or all possible combinations have been exhausted, in which case controls moves to the *Step Failed* state. It should be noted that the selection will also consider subsets containing a single agent, so the process fails only when no agent can successfully perform a step.

Depending on the outcome of the previous stage, either of the rules NotifySuccess or NotifyFailure of the Control API notify the environment of the success or failure of the step, and return to the *Idle* state awaiting further commands from the environment (e.g., another *step* command to continue the computation).

Control API

```
NotifySuccess ≡
    $stepCount$ := $stepCount$ + 1
```

3.2.4 Concurrently Running Agents

We can abstract away from the details of interleaved execution of selected agents in every step of the simulation and model the process in a parallel form. This abstraction is beneficial as it removes the unnecessary sequential order of the execution of agents, hence avoiding over-specification of the engine, and it allows for a more efficient implementation of the engine by *a)* removing the explicit control flow loop around the interpretation of single parse tree nodes (see Figure 3.9) and *b)* enabling concurrent execution of agents on multi-processor machines.

In order to run agent programs in parallel, every function and rule related to the interpretation of the programs should be parameterized by the agents accessing them. As a result, the control state diagram of the scheduler will be reduced to that of Figure 3.10. The RunAgentPrograms rule in the diagram will directly use a parameterized version of the ExecuteTree rule, thereby eliminating the control state diagram of the interpreter.

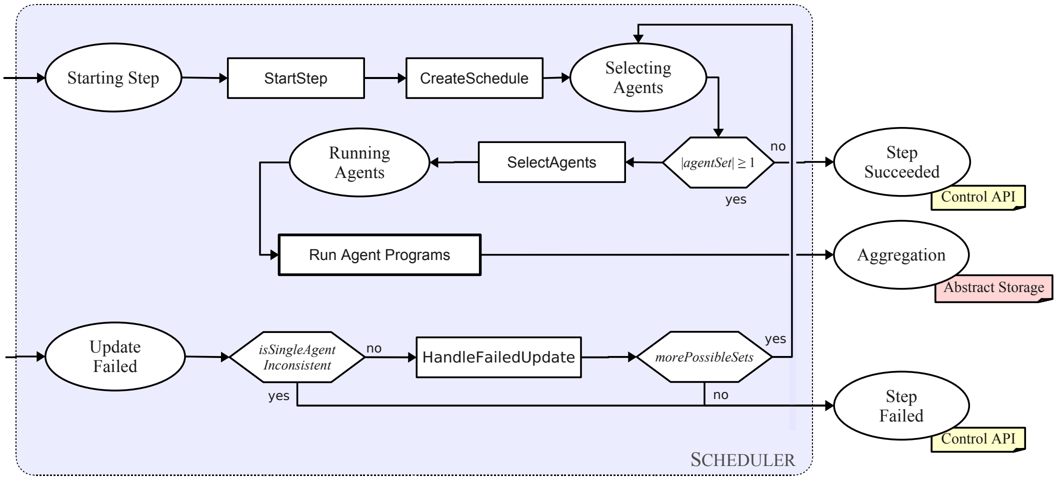


Figure 3.10: Revised Control State ASM of a *step* command: Concurrent Scheduler

Scheduler

```
RunAgentPrograms ≡
  forall $a \in selectedAgentsSet$ do
    let $p$ = $getValue((\text{"program"}, \langle a \rangle))$ in
      seq
        $pos(a) := root(p)$
        ClearTree($p$)
      seq
        while $\neg isEvaluated(root(p))$ do
          ExecuteTree($a$)
      next
        add $updates(root(p))$ to $updateInstructions$
```

3.3 CoreASM Plugins

In keeping with the micro-kernel spirit of CoreASM, most of the functionality of the engine is implemented through plugins to a minimal kernel. In principle, there are three basic dimensions being considered for extending and altering CoreASM by means of plugins, respectively related to: (i) data structures, (ii) control structures, and (iii) the execution model.

i) The possibility of conveniently extending data structures as needed is extensively discussed in the theoretical ASM literature, e.g. in [9, 8], where the concept of *background* refers to an implicitly given part of an abstract machine state, assuming that it provides whatever standard means are normally supposed to be available in a given application context [9]. Plugins extending the data structures of the engine provide all that is needed to define and work with new backgrounds, namely (a) an extension to the parser defining the concrete syntax (operators, literals, static functions, etc.) needed for working with elements of the background; (b) an extension to the abstract storage providing encoding and decoding functions for representing elements of the background for storage purposes, and (c) an extension to the interpreter providing the semantics for all the operations defined in the background. The *Set* plugin, presented in Section 5.3.2, is an example of a background plugin (see Figure 3.1).

ii) Plugins can extend the control structures of CoreASM with respect to both new syntactic constructs that are semantically meaningful and those that only provide syntactic sugar (i.e., the semantics of which could also be expressed by means of in-language transformations). These plugins provide specific rule forms, with the understanding that the execution of a rule always results in a (possibly empty) set of updates. Thus, they include (a) an extension to the parser defining the concrete syntax of the rule form; (b) an extension to the interpreter defining the semantics of the rule form.

iii) Finally, the need for altering or extending the execution model is justified by pragmatic considerations. The execution model refers to dynamic features of CoreASM, including scheduling policies, exception handling, and instrumentation of program execution for analytical purposes. Plugins can alter the execution model of the engine either by providing new scheduling policies to the scheduler, used to determine at each step the next set of agents to execute, or by extending the control sate ASM of the engine. See Section 4.5.5 for more details.

In CoreASM, the kernel (see Figure 3.1) only contains the bare essentials, that is, all that is needed to execute only the most basic ASM. As the state of an ASM machine is defined by functions and universes, the two domains of *functions* and *universes* are included in the kernel. Universes are represented through their characteristic functions, hence *Booleans* are also included in the kernel. As an ASM program is defined by a finite number of rules, the domain of *rules* is also included in the kernel. It should be noted that the kernel includes the above mentioned domains, but not all of the expected corresponding backgrounds. For example, while the domain of Booleans (that is, true and false) is in the kernel, the Boolean algebra ($\wedge$, $\vee$, $\neg$, etc.) is not, and is instead provided through a background plugin. In the same vein, while universes are represented in the kernel through set characteristic functions, the background of finite sets is implemented in a plugin, which provides expression syntax for defining them (see the example in Figure 3.3), as well as an implicit representation for storing sets in the abstract state, and implementations of the various set theoretic operations (e.g., $\in$) that work on such implicit representation.

The kernel includes only two types of rules: assignment and **import**. This particular choice is motivated by the fact that without updates established by assignments there would be no way of specifying how the state should evolve, and that **import** has a special role in introducing new elements to the state. All other rule forms (e.g., **if**, **choose**, **forall**), as well as sub-machine calls and macros, are implemented as plugins in a standard library.

Finally, there is a single scheduling policy implemented in the kernel, namely the pseudo-random selection of an arbitrary set of agents at a time, which is sufficient for multi-agent ASMs where no assumptions are made on the scheduling policy.

As already mentioned, the CoreASM engine is accompanied by a *standard library* of plugins including the most common backgrounds and rule forms (i.e., those defined in [20]), an extension library including a small number of specialized backgrounds and rules, and by a set of specifications for writing new plugins that can easily be integrated in the environment. Extension plugins must be explicitly imported into an ASM specification by an explicit **use** directive.

The plugin framework is further discussed in Section 4.5.

Chapter 4

CoreASM: The Kernel

In this chapter, we look into the details of the CoreASM kernel and its four components. We formally define the interfaces of these components in form of functions and operations (ASM rules). In case of the Abstract Storage, we present the initial structure of simulated *states* in CoreASM and formally define the elements of which it consists of. We then provide a detailed specification of the Interpreter, building on the ExecuteTree rule we presented in Section 3.2. In Section 4.3, we look into the concepts of rules and updates in CoreASM and finally conclude this chapter with an overview of the CoreASM plugin framework.

4.1 The Abstract Storage

Abstract Storage maintains a representation of the current state of the simulated machine in CoreASM. In order to distinguish between the values in the simulated state and the values in our ASM model of the engine, we denote the values of the simulated state as *elements* modeled by the domain ELEMENT. There is a special element in the state that represents the *undefined* value or *undef*. Henceforth, this element is denoted by undef_e .

Elements can belong to different backgrounds, such as Set, Number, Map, and so on. The background of every element is defined by the following function whose default value is "Element" for all elements that do not belong to a particular background:

$$bkg : \textsc{Element} \mapsto \textsc{Name}$$

The kernel also defines a notion of equality on elements which can be extended by plugins providing special backgrounds. For any two elements e_1 and e_2, the notion of equality is defined as:[1]

$$equal(e_1, e_2) \equiv equal_{bkg(e_1)}(e_1, e_2) \vee equal_{bkg(e_2)}(e_2, e_1)$$

[1] Here, the notation $f_x(a_1, \ldots, a_n)$ can be seen as a syntactic sugar for $f(x, a_1, \ldots, a_n)$ and if x is missing, it can be interpreted as $f(\mathit{undef}, a_1, \ldots, a_n)$.

providing that[2]

$$\forall e_1, e_2 \in \textsc{Element} \quad \mathit{equal}_{\mathit{Element}}(e_1, e_2) \equiv e_1 = e_2$$

We model the simulated abstract state as an element of the domain $\textsc{State}$ where every $s \in \textsc{State}$ in principle models a mapping from locations to values (elements). We have:

$$\mathit{content} : \textsc{State} \times \textsc{Location} \mapsto \textsc{Element}$$

During a simulation, the current simulated state is represented by the nullary function state: $\textsc{State}$. Locations are values of the domain $\textsc{Location}$ and each represents a pair of function name and a sequence of arguments:

$$\mathit{name}_{lc} : \textsc{Location} \mapsto \textsc{Name}$$
$$\mathit{args}_{lc} : \textsc{Location} \mapsto \textsc{List}(\textsc{Element})$$

We often denote locations by a pair $(f, \langle a_1, \ldots, a_n \rangle)$ where f is the name of the location and $\langle a_1, \ldots, a_n \rangle$ are the arguments.

In addition to its $\mathit{content}$, a CoreASM state also consists of backgrounds, universes, functions and rules. Before we look into functions and universes, we introduce Boolean elements, the most basic type of elements in the state.

Boolean Elements

We model Boolean elements by values of the domain $\textsc{BooleanElement}$ which has only two elements true_e and false_e, respectively representing Boolean values true and false. The following functions map Boolean elements to Boolean values and vice versa.

$$\mathit{booleanElement} : \textsc{Boolean} \mapsto \textsc{BooleanElement}$$
$$\mathit{booleanValue} : \textsc{BooleanElement} \mapsto \textsc{Boolean}$$

For example, we have:

$$\mathit{booleanElement}(\mathit{true}) = \mathsf{true}_e$$
$$\mathit{booleanValue}(\mathsf{true}_e) = \mathit{true}$$

Equality of Boolean elements are simply defined based on the equality of the Boolean values they represent:

$$\mathit{equal}_{\mathit{Boolean}}(b_1, b_2) \equiv \mathit{booleanValue}(b_1) = \mathit{booleanValue}(b_2)$$

For all $b \in \textsc{BooleanElement}$ we have $\mathit{bkg}(b) =$ "Boolean".

[2] In this equation, $\mathit{Element}$ refers to the background name "Element".

Function Elements

Functions defined in a CoreASM state are modeled by function elements, values of the domain $\textsc{FunctionElement}$. Every CoreASM state holds a mapping of function names to function elements:

$$\begin{aligned} \mathit{stateFunction} &: \textsc{State} \times \textsc{Name} \mapsto \textsc{FunctionElement} \\ \mathit{functions} &: \textsc{State} \mapsto \textsc{Set}(\textsc{FunctionElement}) \\ \mathit{functions}(s) &\equiv \{f \mid f \in \textsc{FunctionElement} \wedge (\exists n \in \textsc{Name}, \mathit{stateFunction}(s, n) = f)\} \end{aligned}$$

Function elements in principle represent a mapping from a sequence of elements (arguments of the function) to an element (the value of the function for those arguments):

$$\mathit{value}_{fe} : \textsc{FunctionElement} \times \textsc{List}(\textsc{Element}) \mapsto \textsc{Element}$$

ASM functions are classified into six categories of *monitored* (or *in*), *controlled*, *shared*, *out*, *static*, and *derived*. Monitored functions, or input functions, are those whose values are only read but never updated by the machine and can only be updated by the environment. Controlled functions, are the opposite; their values can be updated only by the machine and not the environment. Shared functions can be updated and read by both the machine and the environment. The values of out functions can only be updated but never read by the machine; they are intended for output and their values can be read by the environment of the machine. Static functions are constants and their values never change in course of an ASM run. Derived functions can be read by both the machine and the environment, but cannot be updated; their values are defined by a fixed scheme in terms of other functions. In CoreASM, classes of function elements are defined by the following function whose default value is *controlled*:[3]

$$\mathit{class}_{fe} : \textsc{FunctionElement} \mapsto \{\mathit{monitored}, \mathit{controlled}, \mathit{out}, \mathit{static}, \mathit{derived}\}$$

Hence, modifiability of a function element f is defined as follows:

$$\mathit{isModifiable}(f) \equiv \mathit{class}_{fe}(f) \in \{\mathit{controlled}, \mathit{out}\}$$

If a function element is modifiable, its value for a particular sequence of arguments can be assigned by the following rule:

Abstract Storage

```
$\textbf{SetValue}_{\textbf{fe}}(f, args, v) \equiv$
  if $\mathit{isModifiable}(f)$ then
    $\mathit{value}_{fe}(f, \mathit{args}) := v$
```

Every function element f is also a member of $\textsc{Element}$ and $\mathit{bkg}(f)$ = "Function". Finally, two function elements are considered to be equal, if for all the possible argu-

[3] CoreASM does not support *shared* functions at this point.

ments, they hold the same values.[4] For all $f_1, f_2 \in \textsc{FunctionElement}$, we have:[5]

$$equal_{Function}(f_1, f_2) \equiv \ \forall a \in \textsc{List}(\textsc{Element}) \ \ value_{fe}(f_1, a) = value_{fe}(f_2, a)$$

To retrieve the value of a function, the following derived function is defined as part of the interface of Abstract Storage:

$$getValue : \textsc{Location} \mapsto \textsc{Element}$$
$$getValue(l) = \begin{cases} value_{fe}(\mathcal{F}, args_{lc}(l)), & \text{if } value_{fe}(\mathcal{F}, args_{lc}(l)) \neq undef; \\ \mathsf{undef}_e, & \text{otherwise.} \end{cases}$$

where $\mathcal{F} = stateFunction(state, name_{lc}(l))$. The $getValue$ function is later refined in Appendix A.1. In addition, Abstract Storage also provides the following macro rule to set the value of a location in the state:

Abstract Storage

```
SetValue($l, v$) $\equiv$
  let $\mathcal{F} = stateFunction(state, name_{lc}(l))$ in
    if $\mathcal{F} \neq undef$ then
      SetValue$_{\mathsf{fe}}$($\mathcal{F}, args_{lc}(l), v$)
```

Universe Element

Universe elements, values of domain $\textsc{UniverseElement}$, represents the universes defined in a CoreASM state. Every CoreASM state holds a mapping of universe names to universe elements defined in that state:

$$stateUniverse : \textsc{State} \times \textsc{Name} \mapsto \textsc{UniverseElement}$$
$$universes : \textsc{State} \mapsto \textsc{Set}(\textsc{UniverseElement})$$
$$universes(s) \equiv \{u \mid u \in \textsc{UniverseElement} \wedge (\exists n \in \textsc{Name}, stateUniverse(s, n) = u)\}$$

Since universes are sets of elements (or values in ASM), we model them by their set characteristic functions. Hence, every universe element is also a function element. We have:

$$\forall u \in \textsc{UniverseElement}, \ u \in \textsc{FunctionElement}$$

To conveniently view universe elements as sets, we define a membership function on universes:

$$member_{ue} : \textsc{UniverseElement} \times \textsc{Element} \mapsto \textsc{Boolean}$$

[4] Since this definition is not necessarily computable, in practice we assume any two distinct functions to be unequal, unless defined otherwise (e.g., see Section 5.3.7). Hence, we have:

$$\forall f_1, f_2 \in \textsc{FunctionElement} \ \ \textit{equal}_{Function}(f_1, f_2) \equiv f_1 = f_2$$

[5] In ASMs, all functions are total. Partial functions are turned into total functions by introducing a fixed special value *undef* and interpreting $f(x) = $ *undef* as $f(x)$ being undefined. [20]

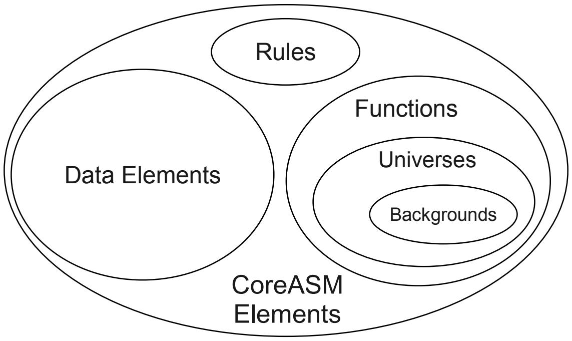


Figure 4.1: CoreASM Elements in the Kernel

For example, if element e belongs to the universe u in the current state of the simulated machine, we have $member_{ue}(u, e) = true$. As a result, for every $u \in$ UniverseElement and every $e \in$ Element, we have

$$value_{fe}(u, e) \equiv booleanElement(member_{ue}(u, e))$$
$$\mathsf{SetValue}_{\mathsf{fe}}(u, \langle e \rangle, b) \equiv member_{ue}(u, e) := booleanValue(b)$$

Equality of universes is de ned as the equality of their characteristic functions:

$$\forall u_1, u_2 \in \textsc{UniverseElement} \quad equal_{Universe}(u_1, u_2) \equiv equal_{Function}(u_1, u_2)$$

For all $u \in$ UniverseElement we have $bkg(u)$ = "Universe".

Background Elements

In CoreASM, backgrounds are special universes with a static membership function. The assumption is that backgrounds contain all the elements they represent; e.g., background of sets represent all the possible sets. In principle, backgrounds represent \types" of elements mostly with internal structures. See, for example, how we de ne the backgrounds of character strings and sets in sections 5.2.3 and 5.3.2.

We model backgrounds by elements of the domain BackgroundElement. For every background element b, $newValue(b)$ must be de ned to return a default element of that background; e.g., an empty set, an empty list, and such. We have:

$$newValue : \textsc{BackgroundElement} \mapsto \textsc{Element}$$
$$\forall\ b \in \textsc{BackgroundElement} \quad class_{fe}(b) = static$$
$$equal_{Background}(b_1, b_2) \equiv equal_{Universe}(b_1, b_2)$$
$$\forall b \in \textsc{BackgroundElement} \quad bkg(b) = \text{"Background"}$$

Rule Elements

ASM rules defined in a CoreASM specification (more precisely, defined in the current state of the simulated machine) are modeled by elements of the domain RuleElement. States of CoreASM hold a mapping of rule names to rule elements defined in those states:

$$\begin{aligned} \mathit{stateRule} &: \textsc{State} \times \textsc{Name} \mapsto \textsc{Rule} \\ \mathit{rules} &: \textsc{State} \mapsto \textsc{Set}(\textsc{Rule}) \\ \mathit{rules}(s) &\equiv \{r \mid r \in \textsc{Rule} \wedge (\exists n \in \textsc{Name}, \mathit{stateRule}(s,n) = r)\} \end{aligned}$$

Every rule element has a name[6], a body (which is a node of the parse tree) and a sequence of parameter names, all defined by the following functions:

$$\begin{aligned} \mathit{name}_{re} &: \textsc{Rule} \mapsto \textsc{Name} \\ \mathit{body} &: \textsc{Rule} \mapsto \textsc{Node} \\ \mathit{param} &: \textsc{Rule} \mapsto \textsc{List}(\textsc{Name}) \end{aligned}$$

The equality of two rules is defined as the equality of their names, program bodies, and list of parameters.

$$\begin{aligned} &\mathit{equal}_{\mathit{Rule}}(r_1, r_2) \equiv \\ &\qquad \mathit{name}_{re}(r_1) = \mathit{name}_{re}(r_2) \wedge \mathit{body}(r_1) = \mathit{body}(r_2) \wedge \mathit{param}(r_1) = \mathit{param}(r_2) \end{aligned}$$

For all $r \in \textsc{Rule}$, we have $\mathit{bkg}(r) =$ "Rule".

Enumerable Elements

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

- $\mathit{enumerable} : \textsc{Element} \mapsto \textsc{Boolean}$
  holds *true* if the element is enumerable. By default, $\mathit{enumerable}(e) = \mathit{false}$ for every element e unless otherwise specified.

- $\mathit{enumerate} : \textsc{Element} \mapsto \textsc{Multiset}(\textsc{Element})$
  provides a collection of elements representing the internal structure of the enumerable element.
  $$\mathit{enumerate}(e) \equiv \mathit{enumerate}_{\mathit{bkg}(e)}(e)$$

- $\mathit{size} : \textsc{Element} \mapsto \textsc{Number}$
  returns the size of this enumerable. For every enumerable element e, we have $\mathit{size}(e) = |\mathit{enumerate}(e)|$.

[6] The names of rule elements, universe elements, and function elements should all be unique in any given CoreASM state.

- $\mathit{contains} : \textsc{Element} \times \textsc{Element} \mapsto \textsc{Boolean}$
  $$\mathit{contains}(e_1, e_2) \equiv \begin{cases} \mathit{true}, & \text{if } \mathit{enumerable}(e_1) \wedge e_2 \in \mathit{enumerate}(e_1) \\ \mathit{false}, & \text{otherwise.} \end{cases}$$

Among the elements we have defined so far, universe elements are enumerable (and so are the background elements). We have:

$$\forall u \in \textsc{UniverseElement} \quad \mathit{enumerable}(u) \wedge \mathit{enumerate}(u) = \{x | \mathit{member}_{ue}(u, x)\}$$

4.2 The Interpreter

The Interpreter evaluates an annotated parse tree and depending on the type of the root node, assigns a value, a location, or a multiset of update instructions to the root of the tree. The Interpreter interacts with the Abstract Storage in order to obtain values from the current state.

In this section we recall the ExecuteTree rule we presented in Section 3.2 and provide further details on the process of evaluating parse tree nodes. More specifically, this section refines the macro rule KernelInterpreter used by ExecuteTree.

4.2.1 Notation

We specify the Interpreter as a collection of rules (some embedded in the kernel, others contributed by plugins) which traverse a parse tree while evaluating values, locations and updates.[7] In order to introduce these rules, we state the following assumptions:

1. Nodes of the parse tree belong to the $\textsc{Node}$ universe and the following functions are defined on nodes:

   - $\mathit{first} : \textsc{Node} \mapsto \textsc{Node}$, $\mathit{next} : \textsc{Node} \mapsto \textsc{Node}$, $\mathit{parent} : \textsc{Node} \mapsto \textsc{Node}$ are static functions that implement tree navigation; by using these functions, the Interpreter can access all the children nodes of a given node, or access its parent (see Figure 3.3 for reference).
   - $\mathit{class} : \textsc{Node} \mapsto \textsc{Class}$ returns the syntactical class of a node (i.e., the name of the corresponding grammar non-terminal class); for example RuleDeclaration .
   - $\mathit{grammarRule} : \textsc{Node} \mapsto \textsc{GrammarRule}$ returns the grammar rule that produced that node.
   - $\mathit{token} : \textsc{Node} \mapsto \textsc{Token}$ returns the syntactical token represented by the node (i.e., either a keyword, an identifier, or a literal value).

[7] This section is a revised and extended version of what we have previously published in [28, Section 3].

- $pattern$: $\textsc{Node} \mapsto \textsc{Pattern}$ returns the symbolic name for the specific grammar pattern corresponding to the node; for example, IfThen symbolically represents the pattern **if** ... **then**.
- $[\![\cdot]\!]$: $\textsc{Node} \mapsto \textsc{Location} \times \textsc{Multiset}(\textsc{Update}) \times \textsc{Element}$ holds the result of the interpretation of a node, given by a triple formed by a location (that is, the l-value of an expression, when it is defined), a multiset of update instructions, and a value (that is, the r-value of an expression)[8]. We access elements and establish properties of such triples through the following derived functions:
  - loc : $\textsc{Node} \mapsto \textsc{Location}$ returns the location (l-value) associated to the given node, i.e. $loc(n) \equiv [\![n]\!] \downarrow 1$.
  - $updates$: $\textsc{Node} \mapsto \textsc{Multiset}(\textsc{Update})$ returns the updates associated to the given node, i.e. $updates(n) \equiv [\![n]\!] \downarrow 2$.
  - $value$: $\textsc{Node} \mapsto \textsc{Element}$ returns the value (r-value) associated to the given node, i.e. $value(n) \equiv [\![n]\!] \downarrow 3$.
  - $evaluated$: $\textsc{Node} \mapsto \textsc{Boolean}$ indicates if a node has been evaluated. We have,

    $$evaluated(n) \equiv [\![n]\!] \neq undef$$
- $plugin$: $\textsc{Node} \mapsto \textsc{Plugin}$ is the plugin associated to a node, that is, the plugin responsible for parsing and evaluation of the node.

2. A special variable pos holds at all times the current position in the tree (i.e., the current node being evaluated).

3. We use a form of pattern matching which allows us to concisely denote complex conditions on the nodes. In particular:

   - we denote with $\boxed{?}$ a generic node;
   - we denote with $\boxed{}$ a generic unevaluated node; as an aid to the reader, we will also use the semantically equivalent $\boxed{e}$, $\boxed{r}$, and $\boxed{l}$ to denote unevaluated nodes whose evaluation is expected to result respectively, in a value (from an expression), a multiset of updates (from a rule), and a location;
   - we denote with x an identifier node;
   - we denote with v (value) an evaluated expression node (that is, a node whose $value$ is not $undef$); we denote with u (update multiset) an evaluated statement node (a node whose $updates$ is not $undef$); we denote with l (location) an evaluated expression for which a location has been computed (a node whose loc is not $undef$). We will at times add subscripts to these

[8] The structure of the triple is intended to be mnemonic, with the l-value in the leftmost and the r-value in the rightmost position in the triple.

variables, or use different names for special cases that will be discussed as appropriate;

- we use prefixed Greek letters to denote positions in the parse tree (typically children of the current node, as denoted by pos) as in **if** ${}^{\alpha}e$ **then** ${}^{\beta}r$ where α and β denote, respectively, the condition node and the then-part node of an if statement;
- rules of the form
  $$\llparenthesis pattern \rrparenthesis \;\rightarrow\; actions$$
  are to be intended as
  $$\textbf{if } conditions \textbf{ then } actions$$
  where the $conditions$ are derived from the pattern according to the conventions above, as more formally specified in Table 4.1; in the action part of such a rule, an unquoted and unbound occurrence of l is to be interpreted as the loc of the corresponding node; an unquoted and unbound occurrence of v is to be interpreted as the $value$ of the corresponding node; an unquoted and unbound occurrence of u as the $updates$ of the corresponding node; and an unquoted and unbound occurrence of x as the $token$ of the corresponding node.

Table 4.2 exemplifies how our compact notation can be translated into actual ASM rules.

4. The value of local variables (e.g., those defined in **import** and **let** rules) is maintained by a global dynamic function of the form $env : \textsc{Token} \mapsto \textsc{Element}$. We have
   $$env(x) \equiv top(envStack(x))$$
   where $envStack$ is a function of the form $envStack : \textsc{Token} \mapsto \textsc{Stack}(\textsc{Element})$ which can be maintained by the following rules:

   Interpreter

   AddEnv$(x, v) \equiv$ Push$(envStack(x), v)$
   RemoveEnv$(x) \equiv$ Pop$(envStack(x))$

Notice that, according to the rule ExecuteTree previously described in Section 3.2, interpreter rules in the kernel or from plugins are only executed when $evaluated(pos)$ does not hold, i.e. when the current node has not been fully evaluated yet. Control moves from node to node either by explicitly assigning values to pos, or by setting $[\![pos]\!]$ to a value that is not $undef$; in which case, control is returned to the parent of pos by the ExecuteTree rule (unless an explicit assignment to pos is also made in the same step). Hence, the general strategy in our rules will be to evaluate all needed subtrees

Abbreviation	**Condition part**	**Action part**
α, β etc.		$\mathit{first}(\mathit{pos})$, $\mathit{next}(\mathit{first}(\mathit{pos}))$, etc.
$^{\alpha}\boxed{?}$	$\mathit{class}(\alpha) \neq \mathsf{Id}$	
$^{\alpha}\boxed{}$	$\mathit{class}(\alpha) \neq \mathsf{Id} \wedge \neg \mathit{evaluated}(\alpha)$	
$^{\alpha}\boxed{e}$, $^{\alpha}\boxed{r}$, $^{\alpha}\boxed{l}$ $\star$	$\mathit{class}(\alpha) \neq \mathsf{Id} \wedge \neg \mathit{evaluated}(\alpha)$	
$^{\alpha}x$	$\mathit{class}(\alpha) = \mathsf{Id}$	$\mathit{token}(\alpha)$
$^{\alpha}v$	$\mathit{value}(\alpha) \neq \mathit{undef}$	$\mathit{value}(\alpha)$
$^{\alpha}u$	$\mathit{updates}(\alpha) \neq \mathit{undef}$	$\mathit{updates}(\alpha)$
$^{\alpha}l$	$\mathit{loc}(\alpha) \neq \mathit{undef}$	$\mathit{loc}(\alpha)$

$\star$ These symbols are semantically equivalent to the $\boxed{}$ symbol; as a visual cue to the reader, the embedded letters express the intended result of evaluation.

Table 4.1: Abbreviations in Syntactic Pattern-matching Rules

Compact notation	Actual rule
$⦇\textbf{if } ^{\alpha}\boxed{e} \textbf{ then } ^{\beta}\boxed{r} ⦈ \rightarrow \mathit{pos} := \alpha$	**let** $\alpha = \mathit{first}(\mathit{pos}), \beta = \mathit{next}(\mathit{first}(\mathit{pos}))$ **in** **if** $\mathit{class}(\mathit{pos}) \neq \mathsf{Id}$ $\wedge\ \mathit{pattern}(\mathit{pos}) = \mathsf{IfThen}$ $\wedge\ \mathit{class}(\alpha) \neq \mathsf{Id}$ $\wedge\ \neg \mathit{evaluated}(\alpha)$ $\wedge\ \mathit{class}(\beta) \neq \mathsf{Id}$ $\wedge\ \neg \mathit{evaluated}(\beta)$ **then** $\mathit{pos} := \mathit{first}(\mathit{pos})$
$⦇\textbf{if } ^{\alpha}v \textbf{ then } ^{\beta}\boxed{r} ⦈ \rightarrow \textbf{ if } v = \mathsf{true}_e \textbf{ then } \ldots$	**let** $\alpha = \mathit{first}(\mathit{pos}), \beta = \mathit{next}(\mathit{first}(\mathit{pos}))$ **in** **if** $\mathit{class}(\mathit{pos}) \neq \mathsf{Id}$ $\wedge\ \mathit{pattern}(\mathit{pos}) = \mathsf{IfThen}$ $\wedge\ \mathit{value}(\alpha) \neq \mathit{undef}$ $\wedge\ \mathit{class}(\beta) \neq \mathsf{Id}$ $\wedge\ \neg \mathit{evaluated}(\beta)$ **then** **if** $\mathit{value}(\alpha) = \mathsf{true}_e$ **then** $\ldots$
$⦇\textbf{if } ^{\alpha}v \textbf{ then } ^{\beta}u ⦈ \rightarrow \ldots$	**let** $\alpha = \mathit{first}(\mathit{pos}), \beta = \mathit{next}(\mathit{first}(\mathit{pos}))$ **in** **if** $\mathit{class}(\mathit{pos}) \neq \mathsf{Id}$ $\wedge\ \mathit{pattern}(\mathit{pos}) = \mathsf{IfThen}$ $\wedge\ \mathit{value}(\alpha) \neq \mathit{undef}$ $\wedge\ \mathit{updates}(\beta) \neq \mathit{undef}$ **then** $\ldots$

Table 4.2: Examples of Pattern Matching Notation Translated into ASM Rules

of a node, if any, by orderly assigning pos accordingly; when all needed subtrees are evaluated, we compute the resulting location, updates or value and assign it to $[\![pos]\!]$, thus implicitly returning control back to our parent. As exemplified in Table 4.2, our notation allows us to clearly visualize this process by the progressive substitution of evaluated u nodes for unevaluated $\boxed{r}$ nodes, and of v or l nodes for unevaluated $\boxed{e}$ nodes. Notice that identifiers do not have to be evaluated, hence we do not need a \boxed" version of x.

4.2.2 Kernel Expression Interpreter

As previously described, the kernel interpreter rules implement the Boolean domain (but not the Boolean algebra), function evaluation and rule call (which share the same syntactic pattern), assignment, and import statement. We present in this section rules that result in values, namely for evaluating literals (true, false, undef) and nullary or n-ary functions.

Literals are simply lifted to their semantic counterparts:

Interpreter: Kernel Expressions

```
$(\!|\,\textbf{true}\,|\!)$     $\rightarrow$   $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
$(\!|\,\textbf{false}\,|\!)$    $\rightarrow$   $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
$(\!|\,\textbf{undef}\,|\!)$    $\rightarrow$   $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
$(\!|\,\textbf{self}\,|\!)$     $\rightarrow$   $[\![pos]\!] := (undef, executingAgent, \mathsf{undef}_e)$
```

Evaluation of identifiers as expressions depends on whether the identifier refers to a local variable or a function. To evaluate an identifier as an expression, the Interpreter first checks the set of in-scope local variables for a possible value for the identifier. If the identifier was not a local variable (i.e., it is not found in the local environment), the Interpreter checks if the identifier refers to a (nullary) function, in which case the Abstract Storage is queried for the value of that function in the current state. If instead the identifier is not defined, the macro HandleUndefinedIdentifier (described later) is called. The rule for n-ary functions is similar, except that the arguments of the function are evaluated first. The formal definition is as follows:

Interpreter: Kernel Expressions

```
$(\!|\,{}^{\alpha}x\,|\!)$   $\rightarrow$   if $env(x) \neq undef$ then
                    $[\![pos]\!] := (undef, undef, env(x))$
                  else
                    if $isFunctionName(x)$ then
                      let $l = (x, \langle\rangle)$ in
                        $[\![pos]\!] := (l, undef, getValue(l))$
                    if $undefinedToken(x)$ then
                      HandleUndefinedIdentifier$(pos, x, \langle\rangle)$
```

```
$(\!| {}^{\alpha}x({}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n) |\!)$   $\rightarrow$   if $\mathit{isFunctionName}(x)$ then
                                            choose $i \in [1..n]$ with $\neg\mathit{evaluated}(\lambda_i)$
                                              $\mathit{pos} := \lambda_i$
                                            ifnone
                                              let $l = (x, \langle \mathit{value}(\lambda_1), \ldots, \mathit{value}(\lambda_n)\rangle)$ in
                                                $[\![\mathit{pos}]\!] := (l, \mathit{undef}, \mathit{getValue}(l))$
                                          if $\mathit{undefinedToken}(x)$ then
                                            HandleUndefinedIdentifier$(\mathit{pos}, x, \langle \lambda_1, \ldots, \lambda_n\rangle)$
where
  $\mathit{undefinedToken}(x) \equiv \neg(\mathit{isFunction}(x) \vee \mathit{isRule}(x) \vee \mathit{isUniverse}(x))$
```

Notice how in the second pattern, the $\boxed{?}$ symbol is used to denote arguments, both unevaluated and evaluated. If x is bound to a function, the rule specifies that all arguments must be evaluated, without any specific order, to determine the location of the node. While there are still unevaluated arguments, the rule sets pos to the node representing an unevaluated argument; as soon as the evaluation of the argument is complete, control returns to the parent node (and thus, again to the same rule), until all arguments are evaluated. At this point (**ifnone** branch), the location and values of the function are computed and stored in $[\![\mathit{pos}]\!]$.

Finally, if the Interpreter encounters an identifier that is not bound to any element of the state, the HandleUndefinedIdentifier rule (see Appendix A.2) will consult all the plugins that are registered to handle undefined identifiers. More specifically, such plugins are asked to evaluate the node with the undefined identifier.[9] If none of the plugins could evaluate the node, KernelHandleUndefIdentifier will be called to create a new function element with a default value of undef$_e$ for the given arguments. This default behavior of the kernel is a “liberal" approach toward type-checking; it allows identifiers to be used without declaration, which is suited for early analysis and specification.

Interpreter: Undefined Identifier

```
**KernelHandleUndefIdentifier**$(\mathit{pos}, x, \mathit{args}) \equiv$
  let $f = \mathit{new}(\textsc{FunctionElement})$ do
    $\mathit{stateFunction}(\mathit{state}, x) := f$
    $[\![\mathit{pos}]\!] := ((x, \mathit{args}), \mathit{undef}, \mathrm{undef}_e)$
```

4.2.3 Kernel Rule Interpreter

Rule plugins provide the execution semantics of rules. Execution of rules results in a multiset of update instructions that is the underlying value for the rule node of the parse tree. As discussed in Section 3.2, accumulated update instructions are used by the Abstract Storage to compute the updates set that will ultimately be applied to the current state to generate the next state.

[9] It is considered an error if more than one plugin evaluate the undefined identifier with different results.

We start with the **skip** rule or the no-operation rule. The semantics of the **skip** rule is simply to produce an empty multiset of updates:

Interpreter: Kernel Rules

$(\!|\, \mathbf{skip} \,|\!) \;\rightarrow\; [\![pos]\!] := (undef, \{\!|\,|\!\}, undef)$

Rule Calls

To evaluate an identifier as a rule, the Interpreter first checks if a rule element is bound to the identifier. If so, the RuleCall macro is called to execute the rule. Notice that in this case, arguments are *not* evaluated prior to calling the rule: in fact, the semantics of rule calls in [20] prescribes that the formal parameter in the body of the rule must be substituted with the entire term that is used as the actual argument, not its value.

Interpreter: Kernel Rules

```
$(\!|\, {}^{\alpha}x \,|\!)$    $\rightarrow$    **if** $isRuleName(x)$ **then**
                                    RuleCall$(ruleValue(x), \langle\rangle)$

$(\!|\, {}^{\alpha}x({}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n) \,|\!)$    $\rightarrow$    **if** $isRuleName(x)$ **then**
                                    RuleCall$(ruleValue(x), \langle\lambda_1, \ldots, \lambda_n\rangle)$
```

Traditionally, rule calls in ASMs have been used in two form: as macros, or as sub-machines. The difference between the two forms is that calling a macro simply means executing its body (possibly with parameter substitution) and collecting the resulting updates, whereas running a submachine results in an entire encapsulated computation of the rule, that is iterated until completion, as defined in [20, Section 4.1.2]. Here, we model macro calls, while the effect of submachine calls can simply be achieved by using the **iterate** construct; see Section 5.1.8 for the specification of the **iterate** construct.

As we have already noted, ASMs differ from many other languages in that *call-by-substitution* is used for parameters instead of the more usual *call-by-value*. In other words, actual parameters are evaluated at the point of use (in the callee) rather than at the point of call (in the caller). Due to the presence of **seq**-rules, the difference can be observable, as parameters can be evaluated in different states. Hence, we have to substitute the whole parse tree denoting an actual parameter (i.e., an expression) for each occurrence of the corresponding formal parameter in the body of the callee. Also, we substitute parameters in a copy of the callee body, to avoid modifying the original definition.

There are several static semantic constraints on valid rule declarations; for example, it is assumed that the formal parameters of a rule are all pairwise distinct, and that the formal parameters are the only freely occurring variables in the body of the rule (see [20], Definition 2.4.18). For simplicity, we do not explicitly check for such conditions in our specification.

The RuleCall routine, defined below, describes how rule calls (possibly with parameters) are handled.

Interpreter: Kernel Rules

```
RuleCall($r$, $args$) $\equiv$
  if $workCopy(pos) = undef$ then
    let $b' = $ CopyTreeSub($body(r), param(r), args$) in
      $workCopy(pos) := b'$
      $parent(b') := pos$
      $pos := b'$
  else
    $[\![pos]\!] := (undef, updates(workCopy(pos)), value(workCopy(pos)))$
    $workCopy(pos) := undef$
```

The rule CopyTreeSub returns a copy of the given parse tree, where every instance of an identifier node in a given sequence (formal parameters) is substituted by a copy of the corresponding parse tree in another sequence (actual parameters). We assume that the elements in the formal parameters list are all distinct (i.e., it is not possible to specify the same name for two different parameters). Also, formal parameters substitution is applied only to occurrences of formal parameters in the original tree passed as argument, and *not* also on the actual parameters themselves. See Appendix A.2 for the definition of CopyTreeSub.

Assignment and Import

The kernel of the CoreASM engine also includes assignment and **import** rules. Assignment is performed as follows:

Interpreter: Kernel Rules

```
$(\!| {}^{\alpha}\boxed{?} := {}^{\beta}\boxed{?} |\!)$   $\rightarrow$   choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
                                  $pos := \tau$
                                ifnone
                                  if $loc(\alpha) \neq undef$ then
                                    if $isModifiable(stateFunction(state, name_{lc}(loc)))$ then
                                      $[\![pos]\!] := (undef, \{\!|\langle loc(\alpha), value(\beta)\rangle|\!\}, undef)$
                                    else
                                      Error(`Cannot update a non-modifiable function')
                                  else
                                    Error(`Cannot update a non-location.')
```

It is worthwhile to remark that the rule above does not syntactically constrain assignment to be performed exclusively to variables or functions: rather, any plugin can contribute new forms of expressions which, as long as they result in a modifiable location (e.g., not a monitored function), are deemed syntactically acceptable in the lhs of an assignment.

The **import** rule is defined as follows:

Interpreter: Kernel Rules

```
$(\!|\, \textbf{import}\ {}^{\alpha}x\ \textbf{do}\ {}^{\beta}\boxed{r}\ |\!)$   $\rightarrow$   $\textbf{let}\ e = new(\textsc{Element})\ \textbf{in}$
                                        $\mathsf{AddEnv}(x, e)$
                                      $pos := \beta$

$(\!|\, \textbf{import}\ {}^{\alpha}x\ \textbf{do}\ {}^{\beta}u\, |\!)$   $\rightarrow$   $\mathsf{RemoveEnv}(x)$
                                      $[\![pos]\!] := (undef, u, undef)$
```

To perform an **import**, a new element is created and it is assigned to the value of the given identifier (x) in the local environment. The rule part $\boxed{r}$ is then evaluated in this new environment by assigning pos to the corresponding node. The identifier is then removed from the local environment when the evaluation of the rule part is complete.

4.2.4 Operators

Although plugins can extend the CoreASM language by introducing (almost) arbitrary expression forms, operators are treated specially in the CoreASM engine. To avoid lengthy expressions with unnecessary parenthesis, the engine provides plugins with a mechanism to declare a precedence level for the operators they contribute.

Precedence level of an operator is defined by a numeric value $p \in [0 \ldots 1000]$, where 1000 is the highest priority. This value should be attached to all operator patterns. The following example introduces a new operator with precedence level 300:

$$(\!|\, \boxed{e} \qquad \boxed{e}\, |\!)_{[300]} \rightarrow \quad \cdots$$

The only operator provided by the kernel is the equality operator (\="). Two values are considered to be equal if they are equal according to at least one of their corresponding backgrounds. In the following rule, the equality functions provided by the backgrounds of the operands are queried to determine the equality:

Interpreter: Kernel Operators

```
$(\!|\, {}^{\alpha}\boxed{?} = {}^{\beta}\boxed{?}\, |\!)_{[600]}$ $\rightarrow$ $\textbf{choose}\ \lambda \in \{\alpha, \beta\}\ \textbf{with}\ \neg evaluated(\lambda)$
                            $pos := \lambda$
                          $\textbf{ifnone}$
                            $\textbf{let}\ e_1 = value(\alpha),\ e_2 = value(\beta)\ \textbf{in}$
                              $\textbf{let}\ b_1 = bkg(e_1),\ b_2 = bkg(e_2)\ \textbf{in}$
                                $\textbf{if}\ equal_{b_1}(e_1, e_2) \vee equal_{b_2}(e_2, e_1)\ \textbf{then}$
                                  $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                $\textbf{else}$
                                  $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
```

4.3 Rules and Updates

- $uiVal : \textsc{Update} \mapsto \textsc{Element}$
  returns the value associated with the given update instruction.

- $uiAction : \textsc{Update} \mapsto \textsc{Action}$
  returns the action associated with the given update instruction.

- $uiAgents : \textsc{Update} \mapsto \textsc{Set}(\textsc{Element})$
  returns the set of agents that produced the given update instruction.

- $aggStatus : \textsc{Update} \times \textsc{Plugin} \mapsto \{successful, failed\}$
  indicates the aggregation status of an update instruction, set by a given aggregator plugin. If an update instruction ui has not been processed by a plugin, $aggStatus(ui)$ is $undef$.

4.3.2 Aggregation of Updates

According to the original ASM definition, after every computation step, location contents are changed by and only by updates. In order to be faithful to that definition, with the introduction of partial updates, we introduce an *aggregation phase* in every computation step that takes place before the application of updates to the state. *Aggregation* is the process of combining all update instructions affecting a single location, into one single update which is called the *resultant update*. The aggregation phase of a CoreASM step performs aggregation on all locations affected by the step and results in a set of regular updates.[12]

Since the CoreASM kernel does not introduce any special update actions other than the one for regular updates, it only defines the framework in which background plugins can provide their background-specific partial updates and their corresponding aggregation algorithms. We say that a plugin is *responsible* for an action, if it is registered to aggregate update instructions of that action. A plugin is said to be *responsible* for aggregation of a given update instruction if the update instruction contains an action for which the plugin is responsible. Finally a plugin is considered to be *responsible* for aggregation of a given regular update if there is an update instruction that operates on the the same location. A plugin that is registered for aggregation of one or more update action is called an *aggregator* plugin.

Recalling the definition of AggregateUpdates on page 39, Abstract Storage calls the following rule in its *Aggregation* control state before firing the updates to the state (see also Figure 3.8):

Abstract Storage

```
AggregateUpdates ≡
  $updateSet \leftarrow \text{Aggregate}(updateInstructions)$
```

The Aggregate method runs the aggregation method of all the aggregator plugins on the update instructions, gathers the resulting updates and returns the compiled

[12] This is also in line with the *integration* phase introduced in [51].

set. When called for aggregation, an aggregator plugin aggregates all update instructions for which it is responsible and flags them as either successful or failed. It is important to note that the order in which plugins are called to perform aggregation should not affect the resultant updates produced. Also note that the failure in aggregation of a single plugin should not fail the aggregation attempt of other plugins.

Abstract Storage

```
**Aggregate**($updates$) $\equiv$
  let $ap = \{a \mid a \in \textsc{Plugin} \wedge aggregator(a)\}$ in
    seq
      forall $p \in ap$ do
        let $R = aggregatorRule(p)$ in
          $resultantUpdates(p, updates) \leftarrow R(updates)$
    next
      result $:= \bigcup_{p \in ap} resultantUpdates(p, updates)$
```

The *resultantUpdates* function is used to collect resultant updates from plugins for a given multiset, and the $aggregatorRule(p)$ function returns the aggregation rule provided by plugin p. Note that a plugin aggregator rule is expected to accept a multiset of update instructions as an argument, and its invocation should cause the return of its resultant updates with the return-result rule syntax as described in [20, Def. 4.1.7].

Plugin Aggregation Consistency

Aggregation algorithms provided by plugins also implicitly define the acceptable semantics of the combination of updates they process. During an aggregation process, a plugin may encounter a situation where the updates and instructions for a given location cannot be aggregated into a regular update. Such a situation may occur, for example, if there are updates or instructions that are semantically inconsistent, such as addition and removal of the same element from a set.

When the aggregation of all updates and instructions affecting a given location are deemed inconsistent, the plugin flags all updates to the location as *failed*.

Abstract Storage

```
**HandleInconsistentAggregation**($loc, updateMset, plugin$) $\equiv$
  forall $ui \in updateMset$ with $uiLoc(ui) = loc$ do
    $aggStatus(ui, plugin) := failed$
```

Although aggregation for a single location may have failed, the aggregation of the rest of the update instructions a plugin is responsible for would continue.

Basic Update Aggregator

Once aggregation of all aggregator plugins have completed successfully, the resultant update set may still have updates with a regular update action that do not need aggregation but are not flagged as processed. The *Basic Update Aggregator* provided by the Kernel plugin (see Section 3.2.1) solves this problem by returning a set of all regular updates for locations which do not require any aggregation and flagging all those updates as *successful*. The basic update aggregator is called by AggregateUpdates along side all aggregator plugins.

Abstract Storage

```
BasicUpdateAggregator($updateMset$) $\equiv$
    seq
      result := {}
    next
      forall $ui \in updateMset$ with $uiAction(ui) = updateAction$ do
        if $\nexists\ ui2 \in updateMset,\ uiLoc(ui) = uiLoc(ui2) \wedge uiAction(u2) \neq updateAction$ then
          add $ui$ to result
          $aggStatus(ui, kernelPlugin)$ := $successful$
```

4.3.3 Composition of Updates

Aggregation as we have described it so far gives semantically acceptable results with basic ASMs. However, for Turbo ASMs, which allow for sequential composition and iteration of ASMs within one single step of the machine, aggregation alone is insufficient. While the sequential composition of ASMs imposes an order between the sets of updates (on a location), it is not always desirable for a Turbo ASM rule to return aggregated resultant updates. On the other hand, update instructions produced by a Turbo ASM rule has to be composed in a form that preserves the sequential semantics of the updates. As an example, consider the following sequential composition, where $s = \{1, 2\}$:

```
seq
  add 5 to s
  add 7 to s
next
  remove 5 from s
  add 6 to s
```

The semantics of this rule is to add 6 and 7 to s. Since this rule may be executed in parallel with other rules that may also modify the set s, it is desirable that the evaluation of this rule does not result in aggregated updates (i.e., a regular update assigning $\{1, 2, 6, 7\}$ to s). On the other hand, there is an explicit order between the update instructions produced by the two parts of this sequence which has to be reflected in the resulting update multiset. As a result, a special *composition* process has to be introduced on update instructions that composes two multisets of update instructions into one multiset with respect to the order of updates. In the above example, removing 5 from s neutralizes the addition of 5 in the first step and so neither of the two modifications will appear in the result of the composition, which will be $\{\!\!\{\langle(\text{“s”}, \langle\rangle), 6, \mathit{setAddAction}\rangle, \langle(\text{“s”}, \langle\rangle), 7, \mathit{setAddAction}\rangle\}\!\!\}$.

Since the CoreASM kernel does not define any special update action, its composition (captured by the Compose rule defined below) basically relies on the composition behaviors provided by background plugins. As a result, every aggregator plugin is required to also provide a composition algorithm which, when given two update multisets, produces composed update instructions for all locations for which the plugin is responsible.

It is important to note that the Compose rule expects the first update multiset to be consistent with respect to typical ASM consistency and aggregation consistency. The result of sequential composition of the two update multisets would then be the union of all composed update instructions produced by individual plugins.

Abstract Storage

```
Compose($uMset_1, uMset_2$) ≡
   seq
     let $ap = \{a \mid a \in \text{PLUGIN} \wedge aggregator(a)\}$ in
        forall $p \in ap$ do
           let $R = composerRule(p)$ in
              $composedUpdates(p, uMset_1, uMset_2) \leftarrow R(uMset_1, uMset_2)$
   next
     result := $\bigcup_{p \in ap} composedUpdates(p, uMset_1, uMset_2)$
```

In the above rule, the $\mathit{composedUpdates}$ function is used to collect the updates resulting from plugins performing sequential composition of two update multisets. The $\mathit{composerRule}$ function is expected to return the composition behavior of the given plugin, implementing the composition of updates on locations for which it is responsible. Note that the composition rule for each plugin is expected to accept two multisets as arguments, and its invocation should cause the return of the sequentially composed update multiset with the return-result rule syntax as described in [20, Def. 4.1.7].

A plugin which provides aggregation, must also provide facilities for sequential composition of actions for which it is responsible. A plugin is deemed responsible for the composition of updates at a given location, if and only if:

- The plugin is responsible for aggregation of the location with respect to the second update multiset.

- The plugin is responsible for aggregation of a location with respect to the first update multiset, if and only if that location is not affected by the second update multiset.

Basic Update Composer

To complement the basic update aggregator we introduced earlier, the Kernel plugin also provides a default update composition behavior. The *Basic Update Composer* is responsible for performing sequential composition of locations affected solely by basic updates. Sequential composition of updates in basic ASMs (without partial updates) is formally defined in [20, Def. 4.1.1] as

$$U \oplus H = \{u \in U \mid location(u) \notin locations(H)\} \cup H$$

In CoreASM, with the existence of partial updates, sequential composition of basic updates is similarly defined as:

$$\begin{aligned} compose(U, H) \equiv\ & \{u \in U \mid location(u) \notin locations(H) \wedge isBasicUpdate(u)\} \\ & \cup \{u \in H \mid isBasicUpdate(u)\} \end{aligned}$$

The basic update composer is then defined as follows:

```
                                                                 Abstract Storage
BasicUpdateComposer($uMset_1, uMset_2$) $\equiv$
  result := $\{ui_1 \mid ui_1 \in uMset_1 \wedge isBasicUpdate(uMset_1, ui1) \wedge \neg locUpdated(uMset_2, uiLoc(ui_1))\}$
            $\cup \{ui_2 \mid ui_2 \in uMset_2 \ \wedge isBasicUpdate(uMset_2, ui_2)\}$
where
  $isBasicUpdate(uMset, ui) \ \equiv\ \forall \langle l, v, a\rangle \in uMset,\ l = uiLoc(ui) \Rightarrow a = updateAction$
  $locUpdated(uMset, l) \ \equiv\ \exists\ ui \in uMset,\ uiLoc(ui) = l$
```

We refer to Mashaal Memon's M.Sc. thesis [64] for further details on aggregation and composition of updates.

4.4 The Parser

CoreASM offers the possibility of extending and modifying the syntax and semantics of its language, keeping only the bare essential parts of the ASM language as static. In order to achieve this goal, CoreASM plugins should be able to extend the grammar of the core language by providing new grammar rules together with their semantics. As a result, the kernel of the engine does not have a comprehensive parser. Plugins used in a given specification can provide portions of the grammar (sets of grammar rules) of the language based on which the specification has to be parsed. Upon loading a specification, the engine will combine all the provided grammar rules into a single grammar. Based on this grammar, a parser is generated which will be used to generate the parse tree of the specification. Hence, the CoreASM parser is in fact a *parser generator* which, when given a grammar, produces a parser that can be used to parse a given specification. As a result, the grammar used for two different specifications may be different, depending on the plugins required by the specifications. One of the challenges in the implementation of CoreASM had been to equip the engine with a fast parser generator capable of generating parsers with look-ahead of more than one to allow co-existence of more than one grammar rule starting with the same pattern.

We do not intend to specify the details of the CoreASM parser; we only require that the parser provides the following function and rule as part of its interface:

- A function of the form $\mathit{requestedPlugins} : \textsc{Specification} \mapsto \textsc{Set}(\textsc{Plugin})$ that for every specification returns the list of plugins used by that specification. In practice, this would be achieved by looking for the **use** clauses in the specification.

- An ASM rule of the form Parse($\mathit{spec}, \mathcal{G}$) that parses the given specification spec with respect to the given grammar $\mathcal{G}$, produces a parse tree of nodes (values of the domain Node, see Section 4.2.1) representing the specification, and returns the root node of the parse tree.

4.5 The Plugin Framework

The CoreASM plugin architecture supports two extension mechanisms: plugins can either extend the functionality of specific components of the engine, by contributing additional data or behavior to those components (i.e., adding new grammar rules to the Parser, new semantic rules to the Interpreter, new backgrounds, universes, and functions to the Abstract Storage, and new policies to the Scheduler) or they can extend the control state ASM of the engine, by interposing their own code in between state transitions.

Practically speaking, a CoreASM plugin can be implemented as a Java class that implements one or more of the interfaces defined by the CoreASM extensibility framework (see Table 4.3 and also Section 6.2.1). In this section we look at various plugin interfaces and explore the mechanisms through which they extend the CoreASM engine.

4.5.1 Parser Extensions

Plugins can implement the *Parser Plugin* interface and/or the *Operator Provider* interface to extend the Parser by respectively contributing additional grammar rules and new operator descriptions. We assume that for any parser plugin pp, $\mathit{pluginGrammar}(pp)$ holds the set of all the grammar rules contributed by pp, and for any operator provider op, $\mathit{pluginOperators}(op)$ holds the descriptions (syntax and semantics) of new operators contributed by op.

Before parsing a specification, the engine gathers all the grammar rules and operator descriptions provided by all parser plugins and operator providers. The Parser then combines these grammar rules and operator descriptions with the kernel grammar and builds a new `parser' to scan the specification. While building the abstract syntax tree, this parser labels the nodes that are created by plugin-provided grammar

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

Table 4.3: CoreASM Plugin Interfaces

rules with the plugin's identifier; these labels can later be used by the Interpreter to evaluate the nodes.

Parser plugins and operator providers are probed by the LoadSpecPlugins rule before the engine starts parsing the specification (see Figure 3.5). This rule iterates over all the plugins required by the loaded specification and after ensuring dependency requirements, loads the plugins by calling the LoadPlugin rule presented below. The latter initializes the plugin, then loads all the provided grammar rules and operator descriptions to be processed by the parser in the next step of the process.

Control API

```
**LoadPlugin**($p$) $\equiv$
  **if** $p \notin loadedPlugins$ **then**
     **seq**
       InitializePlugin($p$)
     **next**
       **add** $p$ **to** $loadedPlugins$
       **if** $isParserPlugin(p)$ **then**
          **add** $pluginGrammar(p)$ **to** $grammarRules$
       **if** $isOperatorProvider(p)$ **then**
          **add** $pluginOperators(p)$ **to** $operatorRules$

**InitializePlugin**($p$) $\equiv$
  **let** $R$ = $pluginInitRule(p)$ **in**
     $R$
```

4.5.2 Interpreter Extensions

Plugins can extend the Interpreter component of the engine by implementing either the *Interpreter Plugin* interface or the *Operator Provider* interface (or both). These plugins provide the semantics for rules and operations contributed as per Section 4.5.1. Traversing the abstract syntax tree, the ExecuteTree rule of the Interpreter (see Figure 3.9) uses these semantic rules to evaluate nodes that correspond to the extended grammar rules.

The semantics contributed by a plugin p which implements the Interpreter Plugin interface can be obtained through $pluginRule(p)$. As already mentioned earlier, nodes of the parse tree corresponding to grammar rules provided by a plugin are annotated with the plugin identifier. If a node is found to refer to a plugin, the Interpreter obtains the semantic rules provided by that plugin and executes it; otherwise, the default kernel Interpreter rules are used (see ExecuteTree on page 39).

A similar approach is also used by the KernelInterpreter 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 [64].

4.5.3 Abstract Storage Extensions

Vocabulary Extender plugins extend the vocabulary of the CoreASM state by contributing new backgrounds, universes, and functions to the Abstract Storage. Such plugins in fact extend the initial state and the signature of the simulated machine. The following functions, defined on vocabulary extender plugins, respectively hold the backgrounds, universes, functions, and rule elements such plugins provide:

$$\begin{aligned}
pluginBackgrounds &: \textsc{Plugin} \mapsto (\textsc{Name} \mapsto \textsc{BackgroundElement})\\
pluginUniverses &: \textsc{Plugin} \mapsto (\textsc{Name} \mapsto \textsc{UniverseElement})\\
pluginFunctions &: \textsc{Plugin} \mapsto (\textsc{Name} \mapsto \textsc{FunctionElement})\\
pluginRules &: \textsc{Plugin} \mapsto (\textsc{Name} \mapsto \textsc{Rule})
\end{aligned}$$

In the Abstract Storage, $stateUniverse$ and $stateFunction$ bind the names of functions and universes in the CoreASM state to the mathematical objects that represent them (see Section 4.1). Backgrounds are considered as special universes and hence are handled by $stateUniverse$. The value of these functions is initialized by the InitAbstractStorage rule (see Figure 3.5). While creating the default universe and functions, the engine calls LoadVocabularyPlugins to iterate over all vocabulary extender plugins and to extend the CoreASM state with the vocabulary they provide.

Abstract Storage

```
**LoadVocabularyPlugins**($state$) $\equiv$
  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
        $stateFunction(state, fName) := f$
      forall $(rName, rBody) \in pluginRules(p)$ do
        $stateRule(state, rName) := rBody$
```

Plugins can also implement the *Aggregator* interface and provide aggregation and composition rules to be applied on update instructions before they are submitted to the state. Aggregator plugins are called to aggregate update instructions by the AggregateUpdate rule in the *Aggregation* state of the engine; see Figure 3.8 and Section 4.3.2 for more details. For any aggregator plugin ap, $aggregatorRule(ap)$ and $composerRule(ap)$ respectively hold the aggregation and composition behaviors provided by ap.

4.5.4 Scheduler Extensions

Policy plugins, also called *Scheduler plugins*, 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. We assume that for any scheduling plugin sp, $pluginSchedulingPolicy(sp)$ holds the scheduling policy provided by sp. For any scheduling policy, the following functions should be defined:

- $newSchedulingGroup : \textsc{SchedulingPolicy} \mapsto \textsc{SchedulingGroup}$
  returns a new scheduling group for the given policy. A scheduling group binds a group of schedules together. The exact semantics of such a group would be defined by the scheduling policy. For example, in a one-by-one scheduling policy that tries to offer a fair schedule, all the schedules created within a group share the same `memory', i.e. they avoid scheduling already scheduled elements before scheduling the `remaining' elements.

- $newScheduleRule : \textsc{SchedulingPolicy} \mapsto \textsc{Rule}$
  returns an ASM rule modeling a function of the form

  $$f : \textsc{SchedulingGroup} \times \textsc{Set} \mapsto \textsc{List}(\textsc{Set})$$

  that given a scheduling group and an initial set of elements (agents), provides a new schedule based on the given policy. The schedule is in form of a list of subsets of the initial set of elements. For example, a schedule on the set $\{a, b, c\}$ can be $\langle\{a, b, c\}, \{a, b\}, \{b, c\}\rangle$ or $\langle\{c\}\rangle$.

See Section 5.4.2 for an example of a policy plugin.

4.5.5 Extension Point Plugins

In addition to modular extensions of specific components, plugins can also extend the control state of the engine by registering themselves for *Extension Points*. Each control state transition in the execution engine is associated to an extension point. At each extension point, if there is any plugin registered for that point, the code contributed by the plugin for that transition is executed before the engine proceeds to the next control state. Such a mechanism enables arbitrary extensions to the engine's lifecycle, 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). A plugin, for example, could monitor the updates that are generated by a step before they are actually applied to the current state of the simulated machine, possibly checking conditions on these updates and thus implementing a kind of watches (i.e., displaying updates to certain locations) or watch-points (i.e., suspending execution of the engine when certain updates are generated), which are useful for debugging purposes. As an additional example, a plugin could provide syntax for declaring assertions and invariants. Assertions have to be checked when the corresponding node is evaluated, hence the plugin would also implement the Interpreter extension to give semantics to assertions. In contrast, invariants have to be checked at each step (not when a particular rule is executed), for example immediately before applying updates: thus, the plugin would hook on the FireUpdateSet extension point to check that the declared invariants really hold in each state.

As we mentioned earlier, we have used a variant of control state ASMs to present a high-level specification of the CoreASM engine. Recalling the definition of control state ASMs from Section 2.3, a control state ASM is an ASM whose rules are all of the form presented in Figure 2.1.

To model the CoreASM engine, we introduce a variation of control state ASMs, called an *Extensible Control State ASM*, which is a control state ASM with an additional (and potentially dynamic) set of *extension point plugins* contributing supplementary rules that are executed before the machine switches to a new state (i.e. before *ctl_state* gets a new value).

Extensible control state ASMs are pictured with almost the same control state diagrams as shown in Figure 2.1. The difference is that in EFSM diagrams, the transition with an extension point is marked with a small diamond;[13] see Figure 4.2(a)

[13] In order not to confuse the reader, we have omitted the diamond from our diagrams. However, this should not be a concern since the extension points are always on the transitions leading to control states.

for an example. Rules of extensible control state ASMs are formulated in textual form by a set of *Extensible Finite State Machine* (EFSM) rules, where EFSM is defined as follows:

EFSM

```
EFSM($i$, if $cond$ then $rule$, $j$) $\equiv$
  if $ctl\_state = i$ and $cond$ then
    $rule$ seq Proceed($i, j$)

Proceed($i, j$) $\equiv$
  seq
    forall $p \in extensionPointPlugins$ do
      $marked(p) := isPluginRegisteredForTransition(p, i, j)$
  seq
    iterate
      let $eps = \{p \mid p \in extensionPointPlugins$ with $marked(p)\}$ in
        choose $p' \in eps$ with $\forall p'' \in eps$ holds $priority(p') \geq priority(p'')$ do
          $marked(p') := false$
          let $R = pluginExtensionRule(p')$ in
            $R(i, j)$
  next
    $ctl\_state := j$
where
  $priority(p) \equiv pluginCallPriority(p, i, j)$
```

An EFSM rule, instead of updating the control state of the machine in parallel with the execution of the transition rule, first executes the transition rule, then iterates over all the extension point plugins (according to their priority) and one by one executes their extension rules before switching the control state of the machine to a new state.[14]

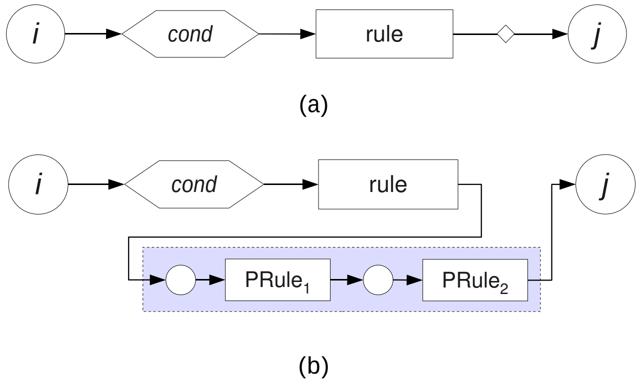
As an example, the extensible control state ASM of Figure 4.2(a) can be executed with a set of extension point plugins $\{p_1, p_2\}$ contributing rules $PRule_1$ and $PRule_2$ which extend the control state of the machine (during its execution) to the control state ASM of Figure 4.2(b).

The following functions are defined on extension point plugins:

- $isPluginRegisteredForTransition$: Plugin × EngineMode × EngineMode $\mapsto$ Boolean
  holds true if the given plugin is registered to extend the behavior of the transition between the two given engine modes.

- $pluginExtensionRule$: Plugin $\mapsto$ Rule
  returns the behavior of the plugin on extension points it is registered for.

[14] If two plugins have the same call priority, their rules will be executed in a non-deterministic order.



(a)

(b)

Figure 4.2: (a) An extensible control state ASM and (b) one of its possible extensions

- $\mathit{pluginCallPriority} : \textsc{Plugin} \times \textsc{EngineMode} \times \textsc{EngineMode} \mapsto \textsc{Number}$
  is the call priority of the plugin on the extension point between the two engine modes. Zero (0) is the lowest priority and 100 is the highest call priority. The engine will consider this priority when calling plugins at extension point transitions. Default call priority is 50.

The *Signature* and *IO* plugins from the standard CoreASM library, among others, implement the Extension Point interface to extend the control state ASM of the engine. We will look into these plugins in more detail in sections 5.4.1 and 5.4.3.

4.5.6 Plugin Service Interface

In many cases, there is a legitimate need for the environment of the CoreASM engine (e.g., the GUI of a simulator or of a debugger) to interact directly with some plugins. To support this interaction, the CoreASM extensibility framework introduces the concept of a *Plugin Service Interface* through which plugins can expose part of their functionality to the environment of the engine.

$$\mathit{pluginServiceInterface} : \textsc{Plugin} \mapsto \textsc{PluginServiceInterface}$$

The Plugin Service Interface allows CoreASM plugins to define and provide their own interfaces to the environment. Applications utilizing the engine can access these interfaces through Control API and directly interact with such plugins. As an example, the IO Plugin provides its own interface to expose the output of its **print** rules to the environment of the engine (see Section 5.4.3). A GUI for the engine, for example, can utilize this interface to obtain the printed output and display it in a console window.

As each plugin exposes different functionalities, users of the Plugin Service Interface have to know in advance what to expect from a specific plugin. This requirement is in keeping with the assumption that the environment will access specific services from a specific plugin, as in the case of **print** rules.

4.5.7 Plugin Background

We model CoreASM plugins by elements of a domain $\textsc{Plugin}$. In addition to the special-purpose functions mentioned in this chapter, the following functions define a general interface for all plugins:

- $\mathit{pluginName} : \textsc{Plugin} \mapsto \textsc{Name}$
  returns the unique name of a plugin. The engine cannot load two plugins that share the same name.
- $\mathit{pluginVersion} : \textsc{Plugin} \mapsto \textsc{Version}$
  returns the version information of the given plugin.
- $\mathit{pluginDependencySet} : \textsc{Plugin} \mapsto \textsc{Set}(\textsc{Name} \times \textsc{Version})$
  is a set of the names and minimum required version of all the plugins that this plugin depends on.
- $\mathit{pluginLoadPriority} : \textsc{Plugin} \mapsto \textsc{Number}$
  returns the suggested loading priority of this plugin. Zero (0) is the lowest priority and 100 is the highest loading priority. The engine will consider this priority when loading plugins. All plugins with the same priority level will be loaded in a non-deterministic order.
- $\mathit{pluginInitRule} : \textsc{Plugin} \mapsto \textsc{Rule}$
  provides an ASM rule that initializes the plugin. This rule is called when the plugin is loaded by the engine; see the LoadPlugin rule on page 66.

For convenience, CoreASM allows plugins to be packaged together in one plugin, called a *package plugin*. For example, a set of standard CoreASM plugins (such as sets, numbers, and lists) can be packed in package plugin called the \Standard Plugin". If a plugin p is a package plugin, the value of $\mathit{isPackagePlugin}(p)$ holds true and $\mathit{enclosedPlugins}(p)$ returns the set of all the plugins enclosed in p.

Chapter 5

CoreASM: The Plugins

Most of the functionalities of CoreASM and its language constructs are provided through plugins to the CoreASM kernel. In this chapter we present the specification of those plugins that are currently available as part the CoreASM project. Most of these plugins are part of the standard library of CoreASM and can be loaded by simply loading the `Standard` package plugin.

Here, we divide the plugins into four categories: plugins that extend the CoreASM language by introducing new rule constructs (Section 5.1), plugins that provide the primitive data types such as numbers and character strings (Section 5.2), plugins that offer more complex data structures as collections of other elements (Section 5.3), and lastly, auxiliary plugins that extend the language and the engine with practically useful constructs and functionalities such as input/output mechanisms and scheduling policies (Section 5.4). The final section of this chapter introduces a special plugin, called JASMine, that allows access to Java objects and classes from CoreASM specifications.

Notation

Throughout this chapter, we use the pattern-action notation of Section 4.2.1 to formally define rule constructs, operators, and expression forms. In addition, we use the notation

```
foo: A -> B
```

in the description of a plugin p, denoting the extension of the vocabulary of the CoreASM state by plugin p through addition of a new Function element *fooFunction*, with the following specification:

$$
\begin{gathered}
\mathit{fooFunction} \in \textsc{FunctionElement} \\
(\text{“foo”}, \mathit{fooFunction}) \in \mathit{pluginFunctions}(p) \\
\mathit{signature}(\mathit{fooFunction}) \equiv \langle \text{“A”}, \text{“B”} \rangle
\end{gathered}
$$

5.1 Standard Rule Constructs

Abstract state machines come with a handful of standard control structures or transition rules (see Section 2.1.3). The most basic ASM rules (assignment, **import**, and **skip**) are defined in the kernel of the CoreASM engine as explained in Section 4.2.3. In this section, we extend the parser and the interpreter of the CoreASM engine through a number of rule plugins that provide the syntax and the semantics of standard and commonly-used ASM rule forms. The result of evaluating each rule, as we explained earlier, will be a multiset of update instructions that becomes the underlying value for the corresponding rule node in the parse tree.

We initiate by presenting rule plugins for all the rule forms defined for basic ASMs; we will then introduce plugins providing Turbo ASMs rule forms.

5.1.1 Block Rule Plugin

The most fundamental control structure in ASM is the block-rule, specified as follows:[1]

Block Rule

$$
\begin{array}{lll}
(\!| \{ {}^{\lambda_1}\boxed{r} \; \ldots {}^{\lambda_n}\boxed{r} \; \} |\!) & \rightarrow & \textbf{choose } i \in [1..n] \textbf{ with } \neg \mathit{evaluated}(\lambda_i) \\
 & & \quad \mathit{pos} := \lambda_i \\
 & & \textbf{ifnone} \\
 & & \quad [\![\mathit{pos}]\!] := (\mathit{undef}, \bigcup_{i \in [1..n]} \mathit{updates}(\lambda_i), \mathit{undef})
\end{array}
$$

Here, all the rules in a block are evaluated in an unspecified order, with the final result being the multiset-union of all the update instructions produced by the various rules in the block.

5.1.2 Conditional Rule Plugin

Close in importance comes the conditional rule construct, or the **if-then-else** rule. We accept a slightly extended syntax, where the guard is not restricted to be a *formula* (basically a Boolean predicate, as per Definition 2.4.14 in [20]), but rather any expression that may return **true**. This guarantees that plugins will be able to extend the set of allowable guards if needed. Notice that this approach is conservative with respect to the standard definition, given that formulae in the sense of [20] are indeed expressions supported by the Predicate Logic plugin (Section 5.2.1) in the CoreASM standard library.

[1] We provide here a rule for an n-elements block, whereas one for a two-elements block would suffice. Notice also that the same rule could be used for the alternative syntax R **par** Q, meaning that P and Q are to be executed in parallel. Finally, also note that we are disregarding here the scope constructors provided by the grammar | either relying on braces { } or on indentation to express nesting are common choices.

Conditional Rule

$$\begin{array}{lcl}
(\!| \textbf{if } {}^{\alpha}\boxed{e} \textbf{ then } {}^{\beta}\boxed{r} \,|\!) & \rightarrow & pos := \alpha \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} \,|\!) & \rightarrow & \textbf{if } v = \mathsf{true}_e \textbf{ then } pos := \beta \textbf{ else } [\![pos]\!] := (undef, \{\!|\!\}, undef) \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}u |\!) & \rightarrow & [\![pos]\!] := (undef, u, undef)
\end{array}$$

$$\begin{array}{lcl}
(\!| \textbf{if } {}^{\alpha}\boxed{e} \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}\boxed{r} \,|\!) & \rightarrow & pos := \alpha \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}\boxed{r} \,|\!) & \rightarrow & \textbf{if } v = \mathsf{true}_e \textbf{ then } pos := \beta \textbf{ else } pos := \gamma \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}u \textbf{ else } {}^{\gamma}\boxed{r} \,|\!) & \rightarrow & [\![pos]\!] := (undef, u, undef) \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}u |\!) & \rightarrow & [\![pos]\!] := (undef, u, undef)
\end{array}$$

5.1.3 The let-rule Plugin

The **let**-rule construct allows the definition of *environment* (read-only) variables (also called *logical* variables) which are not defined in the ASM state, but in a finite local environment. Once defined, the value of a logical variable cannot be updated by a transition rule.

Let Rule

$$\begin{array}{lcl}
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}\boxed{e} \textbf{ in } {}^{\gamma}\boxed{r} \,|\!) & \rightarrow & pos := \beta \\
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}v \textbf{ in } {}^{\gamma}\boxed{r} \,|\!) & \rightarrow & pos := \gamma \\
 & & \mathsf{AddEnv}(x, v) \\
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}v \textbf{ in } {}^{\gamma}u |\!) & \rightarrow & \mathsf{RemoveEnv}(x) \\
 & & [\![pos]\!] := (undef, u, undef)
\end{array}$$

In a **let**-rule of the form `**let** x = e **in** R' the scope of the logical variable x is the rule R but not the expression e.

5.1.4 The extend-rule Plugin

The **extend** rule is a syntactical sugar that imports a new element and adds it to a universe (extends the universe) [20, Table 2.4]. The semantics of an **extend**-rule of the form `**extend** U **with** x **do** R' is as follows: a new element is created and put in a logical variable x, the given rule R is evaluated, and the result of the evaluation of the **extend**-rule will be the union of the update multiset of its inner rule and a single update that adds the new element to universe U.

ExtendRule

$$
\begin{array}{lcl}
\llparenthesis \textbf{extend } {}^{\alpha}\boxed{e}\ \textbf{ with } {}^{\beta}x \textbf{ do}^{\gamma}\boxed{r}\ \rrparenthesis & \rightarrow & pos := \alpha \\
\llparenthesis \textbf{extend } {}^{\alpha}v \textbf{ with } {}^{\beta}x \textbf{ do}^{\gamma}\boxed{r}\ \rrparenthesis & \rightarrow & \textbf{if } isUniverse(v) \textbf{ then} \\
 & & \quad pos := \gamma \\
 & & \quad \textbf{let } e = new(\text{ELEMENT}) \textbf{ in} \\
 & & \qquad \mathsf{AddEnv}(x, e) \\
 & & \textbf{else} \\
 & & \quad \mathsf{Error}(\text{`Extending a non-universe.'}) \\
\llparenthesis \textbf{extend } {}^{\alpha}v \textbf{ with } {}^{\beta}x \textbf{ do}^{\gamma}u \rrparenthesis & \rightarrow & \mathsf{RemoveEnv}(x) \\
 & & \textbf{let } u' = u \cup \{\langle uniLoc(v, e), \mathsf{true}_e, updateAction\rangle\} \textbf{ in} \\
 & & \quad [\![pos]\!] := (undef, u', undef)
\end{array}
$$

where

$uniLoc(v, e) \equiv (name, \langle e\rangle)$ s.t. $stateUniverse(state, name) = v$

5.1.5 The choose-rule Plugin

The **choose**-rule has the form ‘**choose** $x \in X$ **with** φ **do** R' where X is a collection of elements, φ is a Boolean expression and R is a rule. The semantics of the rule is execute R with an arbitrary element x from X that satisfies φ. In CoreASM, we extend this rule form by an optional **ifnone** clause that acts as an ‘else' part: if no such element can be found the **ifnone** rule will be evaluated. We present here a simple form of **choose**-rule, with no additional condition on the chosen value and with an existing **ifnone** clause. A more comprehensive semantic definition is provided in Appendix A.5.1.

Choose Rule

$$
\begin{array}{lcl}
\llparenthesis \textbf{choose } {}^{\alpha}x \textbf{ in } {}^{\beta}\boxed{e}\ \textbf{ do}^{\gamma}\boxed{r}\ \textbf{ ifnone } {}^{\delta}\boxed{r}\ \rrparenthesis & \rightarrow & pos := \beta \\
\llparenthesis \textbf{choose } {}^{\alpha}x \textbf{ in } {}^{\beta}v \textbf{ do}^{\gamma}\boxed{r}\ \textbf{ ifnone } {}^{\delta}\boxed{r}\ \rrparenthesis & \rightarrow & \textbf{if } enumerable(v) \textbf{ then} \\
 & & \quad \textbf{let } s = enumerate(v) \textbf{ in} \\
 & & \qquad \textbf{if } |s| > 0 \textbf{ then} \\
 & & \qquad\quad \textbf{choose } t \in s \textbf{ do} \\
 & & \qquad\qquad \mathsf{AddEnv}(x, t) \\
 & & \qquad\quad pos := \gamma \\
 & & \qquad \textbf{else} \\
 & & \qquad\quad pos := \delta \\
 & & \textbf{else} \\
 & & \quad \mathsf{Error}(\text{`Choosing from a non-enumerable.'}) \\
\llparenthesis \textbf{choose } {}^{\alpha}x \textbf{ in } {}^{\beta}v \textbf{ do}^{\gamma}u \textbf{ ifnone } {}^{\delta}\boxed{r}\ \rrparenthesis & \rightarrow & \mathsf{RemoveEnv}(x) \\
 & & [\![pos]\!] := (undef, u, undef) \\
\llparenthesis \textbf{choose } {}^{\alpha}x \textbf{ in } {}^{\beta}v \textbf{ do}^{\gamma}\boxed{r}\ \textbf{ ifnone } {}^{\delta}u \rrparenthesis & \rightarrow & [\![pos]\!] := (undef, u, undef)
\end{array}
$$

5.1.6 The forall-rule Plugin

The semantic definition of **forall**-rule is similar to that of **choose**-rule with the difference that all the elements of the given enumerable element that satisfy the optional guard are given a chance to be the free variable in the **do**-rule. Here, we present the semantics of **forall**-rule with a guard. The semantics of **forall** with no guard is presented in Appendix A.5.2.

Forall Rule

```
$\llparenthesis \textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}\boxed{e}_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis$   →   $pos$ := $\beta$
                                                         $[\![pos]\!]$ := $(undef, \{\!\!\}, undef)$
                                                         $considered(\beta)$ := $\{\}$
$\llparenthesis \textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis$   →   if $enumerable(v_1)$ then
                                                           let $s$ = $enumerate(v_1) \setminus considered(\beta)$ in
                                                             if $|s| > 0$ then
                                                               choose $t \in s$ do
                                                                 AddEnv($x, t$)
                                                                 $considered(\beta)$ := $considered(\beta) \cup \{t\}$
                                                               $pos$ := $\gamma$
                                                         else
                                                           Error(`Forall on a non-enumerable element')

$\llparenthesis \textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis$   →   if $v_2$ = $\mathsf{true}_e$ then
                                                     $pos$ := $\delta$
                                                   else
                                                     $pos$ := $\beta$
                                                     RemoveEnv($x$)
                                                     ClearTree($\gamma$)
$\llparenthesis \textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}u\ \rrparenthesis$   →   $pos$ := $\beta$
                                                   RemoveEnv($x$)
                                                   ClearTree($\gamma$)
                                                   ClearTree($\delta$)
                                                   $[\![pos]\!]$ := $(undef, updates(pos) \cup u, undef)$
```

Notice that *considered* is used to keep track of values already considered for assignment to the free variable.

5.1.7 The case-rule Plugin

We present here the specification for a plugin implementing a parallel form of a switch case rule. The syntax is similar to the one that is used in [71],[2] but the semantics is quite different. Instead of evaluating the first rule with a matching guard value, all the rules with matching guard values will be evaluated in parallel. In essence, this parallel-case rule acts as a block rule in which all child rules are guarded against a given value.

[2] Here we use colons (:) instead of arrows (→).

To evaluate this rule, the case condition will be evaluated first and then all the guards will be evaluated in an unspecified order. Afterward, rules with a guard value equal to the value of the case condition will be evaluated. Finally, the updates generated by the matching cases are united to form the set of updates generated by the parallel-case rule. Formally, the construct is defined as follows:

Case Rule

```
$\llparenthesis \textbf{case } {}^{\alpha}\boxed{e} \textbf{ of } \{^{\lambda_1}\boxed{e}_1 : {}^{\lambda'_1}\boxed{r}_1 \ldots {}^{\lambda_n}\boxed{e}_n : {}^{\lambda'_n}\boxed{r}_n\} \rrparenthesis \rightarrow$   $pos := \alpha$

$\llparenthesis \textbf{case } {}^{\alpha}v \textbf{ of } \{^{\lambda_1}\boxed{?}_1 : {}^{\lambda'_1}\boxed{r}_1 \ldots {}^{\lambda_n}\boxed{?}_n : {}^{\lambda'_n}\boxed{r}_n\} \rrparenthesis \rightarrow$
                    $\textbf{choose } i \textbf{ in } [1..n] \textbf{ with } \neg evaluated(\lambda_i)$
                        $pos := \lambda_i$

$\llparenthesis \textbf{case } {}^{\alpha}v \textbf{ of } \{^{\lambda_1}v_1 : {}^{\lambda'_1}\boxed{?}_1 \ldots {}^{\lambda_n}v_n : {}^{\lambda'_n}\boxed{?}_n\} \rrparenthesis \rightarrow$
                    $\textbf{choose } i \textbf{ in } [1..n] \textbf{ with } equal(v, v_i) \wedge \neg evaluated(\lambda'_i)$
                        $pos := \lambda'_i$
                    $\textbf{ifnone}$
                        $[\![pos]\!] := (undef, \bigcup_{i \in [1..n] \wedge equal(v,v_i)} updates(\lambda'_i), undef)$
```

5.1.8 The TurboASM Plugin

Basic ASMs are further extended by operators for sequential composition and iteration of ASMs, and also by parameterized submachines [20]. These extended ASMs are called *Turbo ASMs*. Following the definitions of those operators, the TurboASM plugin provides sequentiality and iteration rule forms, together with support for local state definitions and constructs allowing rules to return values.

The seq-rule

Sequential composition of rules is facilitated by the **seq**-rule acting as an operator on rules. According to [20, Def. 4.1.1], the semantics of `P **seq** Q' is defined as the effect of first executing P in the current state $\mathfrak{A}$, and then executing Q in the resulting state $\mathfrak{A} + U_P$ where U_P is the update set produced by P. If U_P is inconsistent, the result of the sequence composition will be U_P.

Since we want to model the effect of evaluating the second rule in a sequence in the state that would be produced by applying the updates produced by the first rule, we have to \simulate" the application of the updates, without really modifying the current state. This is obtained by using a *stack* of states, managed through three macros: PushState copies the current state in the stack, PopState retrieves the state from the top of the stack (thus discarding the current state), and Apply(u) applies the updates in the update set u to the current state. Formal definitions for these macros are given in Appendix A.1. Based on the intuitive understanding of these macros, the interpreter plugin for the **seq**-rule can be specified as follows:

```
                                                        SeqRule
$(\!|{}^{\alpha}\boxed{r}_1\ \mathbf{seq}\ {}^{\beta}\boxed{r}_2|\!)$  →  $pos := \alpha$
$(\!|{}^{\alpha}u_1\ \mathbf{seq}\ {}^{\beta}\boxed{r}_2|\!)$  →  let $uSet$ = Aggregate($u_1$) in
                                if $isConsistent(uSet) \wedge aggregationConsistent(u_1)$ then
                                    PushState
                                    Apply($uSet$)
                                    $pos := \beta$
                                else
                                    $[\![pos]\!] := (undef, u_1, undef)$
$(\!|{}^{\alpha}u_1\ \mathbf{seq}\ {}^{\beta}u_2|\!)$  →  local $uMset$ [$uMset \leftarrow$ Compose($u_1, u_2$)] in
                                PopState
                                $[\![pos]\!] := (undef, uMset, undef)$
```

Before consistency of the update instructions produced by the first rule can be checked, the resultant update instructions must be aggregated into regular updates. If both aggregation consistency and update set consistency hold, the resultant update set is applied to the current state producing a temporary state; otherwise the inconsistent update multiset is returned. If the update instructions produced by the first rule are consistent, the second rule is fired in the temporary state, resulting in the second update multiset. The first and second update multisets must then be sequentially composed. The update multiset resulting from the sequential composition is the update multiset produced by the **seq**-rule in the simulated machine.

In order to improve the readability of specifications, CoreASM provides the following syntax for the sequential composition of rules, in which the **next** keyword is optional:

$$\mathbf{seq}\ P\ \mathbf{next}\ Q\ \equiv\ P\ \mathbf{seq}\ Q$$

The iterate Rule

The **iterate**-rule repeatedly executes its body, until the update set produced is either empty or inconsistent; at that point, the accumulated updates are computed. The resulting update set can be inconsistent if the computation of the last step had produced an inconsistent set of updates. The semantic definition is similar in principle to that of the **seq**-rule:

Iterate Rule

```
$\llparenthesis\,\textbf{iterate}\ {}^{\alpha}\boxed{r}\,\rrparenthesis$   $\rightarrow$   PushState
                               $composedUpdates(pos)$ := $\{\!\mid\mid\!\}$
                               $pos$ := $\alpha$

$\llparenthesis\,\textbf{iterate}\ {}^{\alpha}u\,\rrparenthesis$   $\rightarrow$   if $u \neq \{\!\mid\mid\!\}$ then
                                 let $uSet$ = Aggregate($u$),
                                     $composed \leftarrow$ Compose($composedUpdates(pos), u$) in
                                   $composedUpdates(pos)$ := $composed$
                                   if $aggregationConsistent(u) \wedge isConsistent(uSet)$ then
                                     Apply($uSet$)
                                     ClearTree($\alpha$)
                                     $pos$ := $\alpha$
                                   else
                                     PopState
                                     $[\![pos]\!]$ := $(undef, composed, undef)$
                               else
                                 PopState
                                 $[\![pos]\!]$ := $(undef, composedUpdates(pos), undef)$
```

Notice here how iteration is carried on in a separate state, after saving the original one in the stack. After the iteration is completed, the update instruction multisets are composed into a single multiset of update instructions to be applied to the initial state. The initial state is then restored from the stack, and the computed updates are assigned to the node. Also, notice that after each step in the iteration, the entire subtree is cleared (i.e., the $[\![\cdot]\!]$ function of each node is set to *undef*), so that the computation of the next step can proceed on a clean parse tree.

The while Rule

The non-standard **while**-rule can also be defined in a similar way. The semantics of a rule `**while** (*cond*) R' is to iterate the execution of R as long as *cond* evaluates to true and R does not produce an empty or inconsistent update set. Thus, the following equivalence holds:

$$\textbf{while}\ (cond)\ R\ \equiv\ \textbf{iterate}\ \textbf{if}\ cond\ \textbf{then}\ R$$

Thus, the semantics of the **while** rule closely follows that of the **iterate** rule:

While Rule

```
$\llparenthesis \textbf{while } (^{\alpha}\boxed{e}\,)\ ^{\beta}r \rrparenthesis$   $\rightarrow$   PushState
                                  $composedUpdates(pos)$ := $\{\!|\,|\!\}$
                                  $pos$ := $\alpha$
$\llparenthesis \textbf{while } (^{\alpha}v)\ ^{\beta}\boxed{r} \rrparenthesis$   $\rightarrow$   if $v$ = $\mathsf{true}_e$ then
                                    $pos$ := $\beta$
                                  else
                                    PopState
                                    $[\![pos]\!]$ := $(undef, composedUpdates(pos), undef)$
$\llparenthesis \textbf{while } (^{\alpha}v)\ ^{\beta}u \rrparenthesis$   $\rightarrow$   if $u \neq \{\!|\,|\!\}$ then
                                    let $uSet$ = Aggregate($u$),
                                        $composed \leftarrow$ Compose($composedUpdates(pos), u$) in
                                      $composedUpdates(pos)$ := $composed$
                                      if $aggregationConsistent(u) \land isConsistent(uSet)$ then
                                        Apply($uSet$)
                                        ClearTree($\alpha$)
                                        ClearTree($\beta$)
                                        $pos$ := $\alpha$
                                      else
                                        PopState
                                        $[\![pos]\!]$ := $(undef, composed, undef)$
                                  else
                                    PopState
                                    $[\![pos]\!]$ := $(undef, composedUpdates(pos), undef)$
```

Notice that other choices for the semantics of **while** were also possible: for example, [20, Example 4.1.4] presents a variant that does not terminate when the update set produced by the rule is empty (their Example 4.1.2 is instead consistent with our definition).

More generally, both **iterate** and **while** could also be defined to terminate when the update set contributed by the body of the rule does not modify the state. To our knowledge, this semantics has not been explored and applied in practice.

Local State and Return Values

Local state is introduced in rules by a special syntax [20, Def. 4.1.5] which introduces local state function names together with their initialization rules. Updates made to these special locations are then discarded before returning the final update set to the caller. In the same spirit, return values are simulated by designating a special location in the state, and by using the last update to that location as return value.

We sketch here only the basic idea of how local state and return values are handled. In particular, we omit the details of how local state initialization is performed, based on the observation that a declaration of local state with initialization can be transformed into a declaration without initialization followed by an explicit sequential composition of an assignment and the main rule.

Local Rule

$$\llparenthesis\, \textbf{local}\ {}^{\lambda_1}x_1 \ldots {}^{\lambda_n}x_n\ \textbf{in}\ {}^{\alpha}\boxed{r}\ \rrparenthesis \quad \rightarrow \quad pos := \alpha$$

$$\llparenthesis\, \textbf{local}\ {}^{\lambda_1}x_1 \ldots {}^{\lambda_n}x_n\ \textbf{in}\ {}^{\alpha}u\, \rrparenthesis \quad \rightarrow \quad [\![pos]\!] := (undef, u \ominus \{x_1, \ldots, x_n\}, value(\alpha))$$

where the $\ominus$ operator is defined as follows:

$$U \ominus H = \{\langle l, v, a\rangle \in U \mid name_{lc}(l) \notin H\}$$

A frequent and idiomatic use of Turbo ASMs is to compute functions by executing a rule and then extracting a value from the resulting set of updates, rather than applying the updates to the state. The semantics of the following Turbo ASM call with return values

$$l \leftarrow \mathsf{R}(a_1, \ldots, a_n)$$

is to replace every occurrence of a special variable **result** in the body of the rule R with l, and call rule R [20, Def. 4.1.7]. The following pattern provides a formal semantics for this rule form in CoreASM:

Return Result Rule

$$\llparenthesis\, {}^{\alpha}\boxed{l}\ \leftarrow {}^{\beta}x({}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n)\, \rrparenthesis \quad \rightarrow \quad \begin{array}{l} \textbf{if}\ isRuleName(x)\ \textbf{then} \\ \quad \mathsf{ReturnResultRuleCall}(ruleValue(x), \langle\lambda_1, \ldots, \lambda_n\rangle, l) \end{array}$$

The ReturnResultRuleCall routine, defined below, describes how calls to rules with the special **result** location are handled in CoreASM.

Turbo ASM Plugin

```
**ReturnResultRuleCall**$(r, args, l) \equiv$
  **if** $workCopy(pos) = undef$ **then**
    **let** $params = concat(\text{"result"}, param(r)), args = concat(l, args)$ **in**
      **let** $b' = \mathsf{CopyTreeSub}(body(r), param(r), args)$ **in**
        $workCopy(pos) := b'$
        $parent(b') := pos$
        $pos := b'$
  **else**
    $[\![pos]\!] := (undef, updates(workCopy(pos)), value(workCopy(pos)))$
    $workCopy(pos) := undef$
```

The syntax provided above, however, is not particularly practical, as the computation is restricted to be a statement assigning a value to a given identifier, and so cannot be used inside a complex expression. For example, one has to write

$x \leftarrow R(a_1, \ldots, a_n)$
$y \leftarrow Q(b_1, \ldots, b_m)$
seq
$z := x + y$

instead of the more natural

$$z := R(a_1, \ldots, a_n) + Q(b_1, \ldots, b_m)$$

Hence, we propose here an alternative syntax and semantics of the form

$$\textbf{return } e \textbf{ in } R$$

in which e is an expression and R is a rule. The semantics of this construct is to execute R in the current state $\mathfrak{A}$ and if the resulting update multiset is consistent, evaluate e in the state $\mathfrak{A} + U_R$ (where U_R is the updates produced by R) and return the value of e, discarding U_R. We formally describe this semantics in the following rules:

```
                                                                        ReturnRule
$\llparenthesis \textbf{return } {}^{\alpha}\boxed{e} \textbf{ in } {}^{\beta}r \rrparenthesis$    $\rightarrow$    $pos := \beta$
$\llparenthesis \textbf{return } {}^{\alpha}\boxed{e} \textbf{ in } {}^{\beta}u \rrparenthesis$    $\rightarrow$    $\textbf{let } uSet = \mathsf{Aggregate}(u) \textbf{ in}$
                                      $\textbf{if } isConsistent(uSet) \wedge aggregationConsistent(u) \textbf{ then}$
                                        PushState
                                        Apply($uSet$)
                                        $pos := \alpha$
                                      $\textbf{else}$
                                        $\llbracket pos \rrbracket := (undef, \{\!\!|\,|\!\!\}, \mathsf{undef}_e)$
$\llparenthesis \textbf{return } {}^{\alpha}v \textbf{ in } {}^{\beta}u \rrparenthesis$    $\rightarrow$    PopState
                                    $\llbracket pos \rrbracket := (undef, \{\!\!|\,|\!\!\}, v)$
```

In this construct, the rule r is executed first; the return expression is evaluated in the state obtained by provisionally applying the updates from r to the current state, and the resulting value is returned, while the updates and the provisional state itself are discarded.

5.2 Primitive Data Types

In this section we introduce those plugins that extend the CoreASM engine with backgrounds of primitive data types, basically numbers and character strings. We also include in this section the Predicate Logic plugin that offers Boolean operators defined on Boolean elements introduced in the CoreASM kernel.

5.2.1 The Predicate Logic Plugin

The Predicate Logic plugin provides operators implementing a Boolean algebra. Since the corresponding background is already provided by the kernel, this plugin extends only the parser and the interpreter of the CoreASM engine to provide the standard Boolean operators together with the universal and the existential quantifiers.

The only unary operator provided by this plugin is the negation operator: **not**. The semantics of this operator is very simple and is formally defined by the following rule:

Predicate Logic Plugin: not

```
$\llparenthesis$ not ${}^{\alpha}\boxed{?}\rrparenthesis_{[850]}$ $\rightarrow$ if $\neg evaluated(\alpha)$ then
                                     $pos := \alpha$
                                   else
                                     if $isBoolean(value(\alpha))$ then
                                       if $value(\alpha) = \mathsf{true}_e$ then
                                         $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
                                       else
                                         $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                     if $value(\alpha) = \mathsf{undef}_e$ then
                                       $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

The Predicate Logic plugin also provides the standard binary operators **and**, **or**, **xor**, and **implies**, together with the not-equality operator `!=`. As an example, we present here the semantic definition of the logical implication operator:

Predicate Logic Plugin: implies

```
$\llparenthesis {}^{\alpha}\boxed{?}$ implies ${}^{\beta}\boxed{?}\rrparenthesis_{[375]}$ $\rightarrow$ choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                                   $pos := \lambda$
                                                 ifnone
                                                   if $isBoolean(l) \wedge isBoolean(r)$ then
                                                     if $((value(\alpha) = \mathsf{false}_e) \vee (value(\beta) = \mathsf{true}_e))$ then
                                                       $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                                     else
                                                       $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
                                                   else
                                                     if $\forall x \in \{l, r\}\ isBoolean(x) \vee x = \mathsf{undef}_e$ then
                                                       $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
                                                 where
                                                   $l \equiv value(\alpha), r \equiv value(\beta)$
```

In addition, the Predicate Logic plugin also provides the membership operator `$\in$'. If the operand on the right hand side (rhs) is an enumerable, this operator returns true if that enumerable includes the operand on the left hand side (lhs). We have:

Predicate Logic Plugin: memberof

```
$\llparenthesis {}^{\alpha}\boxed{?}$ memberof ${}^{\beta}\boxed{?}\rrparenthesis_{[550]}$ $\rightarrow$ choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                                    $pos := \lambda$
                                                  ifnone
                                                    if $enumerable(value(\alpha))$ then
                                                      if $value(\beta) \in enumerate(value(\alpha))$ then
                                                        $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                                      else
                                                        $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
                                                    if $value(\alpha) = \mathsf{undef}_e$ then
                                                      $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

The formal definition of other operators is available in Appendix A.5.3.

Two logical quantifiers $\exists$ and $\forall$ are also provided by the Predicate Logic plugin with the following syntax

$$\textbf{exists } x \textbf{ in } X \textbf{ with } \varphi$$
$$\textbf{forall } x \textbf{ in } X \textbf{ holds } \varphi$$

in which X is an enumerable, φ is a Boolean predicate and the scope of x is limited to φ. We present here the semantic definition of the existential quantifier. The definition of the universal quantifier is very similar and is presented in Appendix A.5.3. Notice again the use of the *considered* function to keep track of the elements that we considered so far.

Predicate Logic Plugin: exists

```
$(\!| \textbf{exists}^{\alpha} x \textbf{ in } {}^{\beta}\boxed{e} \textbf{ with } {}^{\gamma}\boxed{e} |\!)$  →  $pos := \beta$
                                                $considered(\beta) := \{\}$
$(\!| \textbf{exists}^{\alpha} x \textbf{ in } {}^{\beta}v \textbf{ with } {}^{\gamma}\boxed{e} |\!)$  →  if $enumerable(v)$ then
                                                  let $s = enumerate(v) \backslash considered(\beta)$ in
                                                    if $|s| > 0$ then
                                                      choose $t \in s$ do
                                                        AddEnv($x, t$)
                                                        $considered(\beta) := considered(\beta) \cup \{t\}$
                                                      $pos := \gamma$
                                                    else
                                                      $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
                                                else
                                                  Error(`Cannot enumerate a non-enumerable element')

$(\!| \textbf{exists}^{\alpha} x \textbf{ in } {}^{\beta}v \textbf{ with } {}^{\gamma}v |\!)$  →  if ($value(\gamma) = \mathsf{true}_e$) then
                                              $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                            else
                                              $pos := \beta$
                                            RemoveEnv($x$)
                                            ClearTree($\gamma$)
```

5.2.2 The Number Plugin

The Number plugin extends the abstract storage, the parser, and the interpreter of the CoreASM engine to provide the *Number* background, representing the domain of Real numbers $\mathbb{R}$, together with necessary functions and operators needed to work with both integer and real numbers. The background of Number elements is defined as $numberBkg \in \textsc{BackgroundElement}$; we have

$$name(numberBkg) = \text{“NUMBER”}$$
$$newValue(numberBkg) = zero$$

Number elements are values of the domain $\textsc{NumberElement}$. We have

$$\forall ne \in \textsc{NumberElement} \quad member_{ue}(numberBkg, n) = true$$

We define the following functions to provide a mapping from Number elements to the actual numeric values they represent and vice versa:

$$\begin{gathered} \mathit{numberElement} : \mathbb{R} \mapsto \textsc{NumberElement} \\ \mathit{numericValue} : \textsc{NumberElement} \mapsto \mathbb{R} \end{gathered}$$

Finally, the equality of two Number elements is defined as the equality of the numeric values they represent (see also Section 4.1):

$$\forall ne' \in \textsc{NumberElement} \quad \mathit{equal}_{\mathit{Number}}(ne, ne') \equiv \mathit{numericValue}(ne) = \mathit{numericValue}(ne')$$

Operators

The Number plugin provides the following numeric operators:

- "+" : the addition binary operator (precedence level: 750)
- "-" : the subtraction binary operator (precedence level: 750)
- "-" : the negation unary operator (precedence level: 850)
- "*" : the multiplication binary operator (precedence level: 800)
- "/" : the division binary operator (precedence level: 800)
- "div" : the integer division binary operator (precedence level: 800)
  $a \textbf{ div } b \equiv \mathit{floor}(a/b)$
- "%" : the modulus (remainder) binary operator (precedence level: 800)
  $a \ \% \ b \equiv \mathit{floor}(a/b)$
- "^" : the exponential binary operator (precedence level: 820)

We present here the semantics of the addition operator (i.e., "+"). The same approach is used to define the rest of the above operators.

Number Plugin

```
$(\!| {}^{\alpha}\boxed{?} + {}^{\beta}\boxed{?} |\!)_{[750]}$    $\rightarrow$    choose $\lambda \in \{\alpha, \beta\}$ with $\neg \mathit{evaluated}(\lambda)$
                                     $\mathit{pos} := \lambda$
                                   ifnone
                                     if $\forall x \in \{l, r\}\ x \in \textsc{NumberElement} \lor x = \mathsf{undef}_e$ then
                                       if $l = \mathsf{undef}_e \lor r = \mathsf{undef}_e$ then
                                         $[\![\mathit{pos}]\!] := (\mathit{undef}, \mathit{undef}, \mathsf{undef}_e)$
                                       else
                                         $[\![\mathit{pos}]\!] := (\mathit{undef}, \mathit{undef}, \mathit{result})$
                                   where
                                     $\mathit{result} \equiv \mathit{numberElement}(\mathit{numericValue}(l) + \mathit{numericValue}(r))$
                                     $l \equiv \mathit{value}(\alpha)$
                                     $r \equiv \mathit{value}(\beta)$
```

The Number plugin also provides the following relational operators defined on Number elements:

- ">" : greater-than binary operator (precedence level: 650)
- ">=" : greater-than or equal-to binary operator (precedence level: 650)
- "<" : less-than binary operator (precedence level: 650)
- "<=" : less-than or equal-to binary operator (precedence level: 650)

The greater-than operator is defined as follows:

Number Plugin

```
$(\!| {}^{\alpha}\boxed{?} > {}^{\beta}\boxed{?} |\!)_{[650]}$   $\rightarrow$   choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                      $pos$ := $\lambda$
                                    ifnone
                                      if $\forall x \in \{l, r\}$ $x \in$ NUMBERELEMENT $\vee$ $x$ = $\mathsf{undef}_e$ then
                                        if $l$ = $\mathsf{undef}_e \vee r$ = $\mathsf{undef}_e$ then
                                          $[\![pos]\!]$ := $(undef, undef, \mathsf{undef}_e)$
                                        else
                                          $[\![pos]\!]$ := $(undef, undef, result)$
                                    where
                                      $result \equiv booleanValue(numericValue(l) > numericValue(r))$
                                      $l \equiv value(\alpha)$
                                      $r \equiv value(\beta)$
```

The semantics of the other three relational operators are also defined in a similar fashion.

Functions

The Number plugin extends the vocabulary of the state with the following two functions:

- `infinity: -> NUMBER`
  returns the positive infinity.
- `toNumber: ELEMENT -> NUMBER`
  if possible, maps the given element to a Number element it represents.

Number Classes

The Number plugin provides the user with the following predicates in order to identify whether a number belongs to a particular numerical class:

- `isNaturalNumber: NUMBER -> BOOLEAN`
  $$fGetValue(isNaturalNumberFunction, \langle n \rangle) = \begin{cases} \mathsf{true}_e, & \text{if } numericValue(n) \in \mathbb{N}; \\ \mathsf{false}_e, & \text{otherwise.} \end{cases} .$$

- `isIntegerNumber: NUMBER -> BOOLEAN`
  $fGetValue(isIntegerNumberFunction, \langle n \rangle) = \begin{cases} \mathsf{true}_e, & \text{if } numericValue(n) \in \mathbb{Z}; \\ \mathsf{false}_e, & \text{otherwise.} \end{cases}$.
- `isRealNumber: NUMBER -> BOOLEAN`
  $fGetValue(isRealNumberFunction, \langle n \rangle) = \begin{cases} \mathsf{true}_e, & \text{if } numericValue(n) \in \mathbb{R}; \\ \mathsf{false}_e, & \text{otherwise.} \end{cases}$.

Number Characteristics

To identify the characteristics of numbers, the following predicates are defined on all Number elements:

- `isEvenNumber: NUMBER -> BOOLEAN`
  $fGetValue(isEvenNumberFunction, \langle n \rangle) =$
  $\begin{cases} \mathsf{true}_e, & \text{if } numericValue(n) \in \mathbb{Z} \ \wedge \ numericValue(n)\%2 = 0; \\ \mathsf{false}_e, & \text{otherwise.} \end{cases}$.
- `isOddNumber: NUMBER -> BOOLEAN`
  $fGetValue(isOddNumberFunction, \langle n \rangle) =$
  $\begin{cases} \mathsf{true}_e, & \text{if } numericValue(n) \in \mathbb{Z} \ \wedge \ numericValue(n)\%2 = 1; \\ \mathsf{false}_e, & \text{otherwise.} \end{cases}$.

Number Ranges

Number plugin also provides the NUMBERRANGE background which is the background of number ranges of the form $[a..b : s]$ where a and b are respectively the starting and the ending values of the range (inclusive) and s is the *step* of the range. The background of Number Range elements is provided by $numberRangeBkg \in$ BACKGROUNDELEMENT, where

$$name(numberRangeBkg) = \text{“NUMBER\_RANGE”}$$
$$newValue(numberRangeBkg) = [0..1 : 1]$$

The following functions are defined on Number Range elements (see Section 4.1):

- $bkg(r)$ = “NumberRange” where $r \in$ NUMBERRANGE.
- $rangeFrom$: NUMBERRANGE ↦ NUMBER
  holds the lower boundary of the Number Range element.
- $rangeTo$: NUMBERRANGE ↦ NUMBER
  holds the upper boundary of the Number Range element.
- $rangeStep$: NUMBERRANGE ↦ NUMBER
  holds the range step.

- $\forall nr_1, nr_2 \in \textsc{NumberRange} \quad \mathit{equal}_{\mathit{NumberRange}}(nr_1, nr_2) \equiv$
  $\mathit{rangeFrom}(nr_1) = \mathit{rangeFrom}(nr_2)$
  $\wedge\ \mathit{rangeTo}(nr_1) = \mathit{rangeTo}(nr_2)$
  $\wedge\ \mathit{rangeStep}(nr_1) = \mathit{rangeStep}(nr_2)$

- $\forall r \in \textsc{NumberRange},\ \mathit{enumerable}(r)$
  All Number Range elements are enumerable.

- $\mathit{enumerate}_{\mathit{IntegerRange}} : \textsc{NumberRange} \mapsto \textsc{List}(\textsc{Element})$
  provides a collection of Elements representing the numbers that are included in the given Number Range.

  $\mathit{enumerate}(r) \equiv [\, x \mid x = \mathit{rangeFrom}(r) + i * \mathit{rangeStep}(r) \wedge i \in \mathbb{N} \wedge x \leq \mathit{rangeTo}(r)]$

The following expression form creates a Number Range element:

Integer Range

```
$(\!|\,[^{\alpha}\boxed{?}\ ..^{\beta}\boxed{?}\ :\ ^{\gamma}\boxed{?}]\,|\!)$   $\rightarrow$   choose $\lambda \in \{\alpha, \beta, \gamma\}$ with $\neg evaluated(\lambda)$
                                     $pos := \lambda$
                                 ifnone
                                     if $\forall v \in \{l, r, s\}\ \ isNumber(v)$ then
                                         let $newRange = newValue(numberRangeBack)$ in
                                             $rangeFrom(newRange) := numericValue(l)$
                                             $rangeTo(newRange) := numericValue(r)$
                                             $rangeStep(newRange) := numericValue(s)$
                                             $[\![pos]\!] := (undef, undef, newRange)$
                                     else
                                         Error(`Both operands must be numbers.')
                                 where
                                     $l \equiv value(\alpha)$
                                     $r \equiv value(\beta)$
                                     $s \equiv value(\gamma)$
```

In the above form, the step of a range (γ) can be omitted in which case it would be considered to be 1.

5.2.3 The String Plugin

The String plugin provides all that is needed to work with character strings as elements of the CoreASM state. The background of String elements is provided by $\mathit{stringBack} \in \textsc{BackgroundElement}$; we have

$$\mathit{name}(\mathit{stringBack}) = \text{“STRING”}$$
$$\mathit{newValue}(\mathit{stringBack}) = \mathit{emptyString}$$

We model String elements as values of a domain $\textsc{StringElement}$. The following functions are defined on String elements:

- $\mathit{stringValue} : \textsc{StringElement} \mapsto \textsc{List}(\textsc{Character})$
  for every String element returns the sequence of characters in that string.

- $\mathit{stringElement} : \textsc{Element} \mapsto \textsc{StringElement}$
  maps every element to a String representation of that element. The exact semantics of this function depends on the Element itself and it is left abstract here.

- $\mathit{concatString} : \textsc{StringElement} \times \textsc{StringElement} \mapsto \textsc{StringElement}$
  concatenates two string elements into one. For all $s_1, s_2 \in \textsc{StringElement}$, we have

$$\mathit{concatString}(s_1, s_2) \equiv \mathit{concat}(\mathit{stringValue}(s_1), \mathit{stringValue}(s_2))$$

For every $s \in \textsc{StringElement}$ we have (see Section

Functions

The String plugin extends the CoreASM state with the following two functions defined on String elements:

- `toString: ELEMENT -> STRING`
  returns a string representation of the given element. We have,
  $\forall e \in \textsc{Element} \quad \mathit{value}_{fe}(\mathit{toStringFunction}, \langle e \rangle) = \mathit{stringElement}(e)$

- `strlen: STRING -> NUMBER`
  returns the length of the given string. For all $s \in \textsc{StringElement}$ we have,
  $\mathit{value}_{fe}(\mathit{strlenFunction}, \langle s \rangle) = \mathit{numberElement}(|\mathit{stringValue}(s)|)$

The String plugin relies on the availability of the Number background provided by the Number plugin.

5.3 Collections

We use the term *collection* to refer to the most abstract concept of a grouping of zero or more elements with potential multiplicities of more than one. In this section, we introduce those CoreASM plugins that offer backgrounds implementing different kinds of collections. The most liberal implementation of collections in CoreASM is provided by the Bag plugin (Section 5.3.3). Other plugins, such as the Set plugin (Section 5.3.2) and the List plugin (Section 5.3.4), offer more specialized forms of collections. The Collection plugin, introduced in Section 5.3.1, provides the foundation for collection backgrounds in CoreASM.

5.3.1 The Collection Plugin

The Collection plugin provides a cornerstone for collections in CoreASM, offering a set of common functions and rule forms defined on collections. However, each specific collection background (e.g., list or set) is provided separately by its corresponding plugin.

Abstract Map Elements

Some collection elements can be represented as a mapping of elements. A collection element that can represent itself as a map is considered to be an AbstractMapElement by the Collections plugin. The value of the following function has to be defined by the background of elements that belong to AbstractMapElement:

$$\mathit{getMap}_{bkg} : \textsc{AbstractMapElement} \mapsto (\textsc{Element} \mapsto \textsc{Element})$$

where bkg is the background of the element.

Modifiable Collections

The Collection plugin introduces a modifiable-collection attribute on elements, defined by the following function:

$$\mathit{isModifiableCollection} : \textsc{Element} \mapsto \textsc{Boolean}$$

The modifiability attribute set on an element indicates that generic collection modifications (at this point limited to addition and removal of an element) can be applied to the element. Plugins that provide modifiable collection elements (such as sets and list) must also provide the semantics of such modifications through two functions of the form

$$\mathit{computeAddUpdate}_{bkg} : \textsc{Location} \times \textsc{Element} \mapsto \textsc{Multiset}(\textsc{Update})$$
$$\mathit{computeRemoveUpdate}_{bkg} : \textsc{Location} \times \textsc{Element} \mapsto \textsc{Multiset}(\textsc{Update})$$

where bkg is the collection background the plugin provides. These two functions are expected to produce proper update instructions to add/remove elements to/from locations holding collection elements.

Rule Forms

The Collection plugin extends the CoreASM language with two rule forms for adding and removing elements to and from collections. As explained above, the semantics of these rule forms relies on the add and remove semantics provided by the plugin of each collection element.

Collection Plugin: Add-To

```
$(\!|\ \textbf{add}\ {}^{\alpha}\boxed{e}\ \textbf{to}\ {}^{\beta}\boxed{l}\ |\!) \rightarrow$  choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
                      $pos := \tau$
                    ifnone
                      let $c = value(\beta)$ in
                        if $isModifiableCollection(c)$ then
                          let $u = computeAddUpdate_{bkg(c)}(loc(\beta), value(\alpha))$ in
                            $[\![pos]\!] := (undef, u, undef)$
```

Collection Plugin: Remove-From

```
$(\!|\ \textbf{remove}\ {}^{\alpha}\boxed{e}\ \textbf{from}\ {}^{\beta}\boxed{l}\ |\!) \rightarrow$  choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
                      $pos := \tau$
                    ifnone
                      let $c = value(\beta)$ in
                        if $isModifiableCollection(c)$ then
                          let $u = computeRemoveUpdate_{bkg(c)}(loc(\beta), value(\alpha))$ in
                            $[\![pos]\!] := (undef, u, undef)$
```

Functions

The Collection plugin also provides the following functions defined on enumerable elements:

- `foldl: ELEMENT * FUNCTION * ELEMENT -> ELEMENT`
  which implements the following function:
  $\mathit{foldl}([x_1, \ldots, x_n], f, i) \equiv f(x_n, f(x_{n-1}, \ldots f(x_1, i))) \ldots)$
- `foldr: ELEMENT * FUNCTION * ELEMENT -> ELEMENT`
  which implements the following function:
  $\mathit{foldr}([x_1, \ldots, x_n], f, i) \equiv f(x_1, f(x_2, \ldots f(x_n, i))) \ldots)$
- `fold: ELEMENT * FUNCTION * ELEMENT -> ELEMENT`
  is the same as `foldr`.
- `fold: ELEMENT * FUNCTION -> ELEMENT`
  which implements the following function:
  $\mathit{map}([x_1, \ldots, x_n], f) \equiv [f(x_1), f(x_2), \ldots f(x_n)]$
- `filter: ELEMENT * FUNCTION -> ELEMENT`
  which implements the following function:
  $\mathit{filter}(\{x_1, \ldots, x_n\}, f) \equiv \{x_i \mid f(x_i)\}$
  $\mathit{filter}([x_1, \ldots, x_n], f) \equiv [x_i \mid f(x_i)]$

The Collection plugin depends on the availability of the Number background provided by the Number plugin.

5.3.2 The Set Plugin

The Set plugin extends the CoreASM state by providing the background of sets with its operations and functions.[3] The background of Set elements is provided by $\mathit{setBack} \in$ BackgroundElement; we have

$$\begin{aligned} \mathit{name}(\mathit{setBack}) &= \text{“SET”} \\ \mathit{newValue}(\mathit{setBack}) &= \mathit{emptySet} \end{aligned}$$

Set elements are values of the domain SetElement. The following functions define the interface of Set elements by providing a mapping between Set elements and the actual set of elements they represent:

- $\mathit{setElement} : \textsc{Set}(\textsc{Element}) \mapsto \textsc{SetElement}$
  for every set of elements, returns a Set element representation of that set.

[3] This section is based on Mashaal Memon's M.Sc. work previously published in [64] with improvements and modifications.

- $setMembers : \textsc{SetElement} \mapsto \textsc{Set}(\textsc{Element})$
  for every Set element, returns the set of its members.

For all $s \in \textsc{SetElement}$ we have:

- $bkg(s)$:= "Set"
- $\forall s' \in \textsc{SetElement} \quad equal_{Set}(s, s') \equiv setMembers(s) = setMembers(s')$
- $enumerable(s)$
  All Set elements are enumerable.
- $enumerate_{Set}(s) = setMembers(s)$.
- $s \in \textsc{FunctionElement}$
  All Set elements also behave as functions.
- $class_{fe}(s) = static$
- $\forall e \in \textsc{Element} \quad value_{fe}(s, \langle e \rangle) \equiv booleanValue(e \in setMembers(s))$
- $s \in \textsc{AbstractMapElement}$
  All Set elements are abstract map elements.
- $getMap_{Set}(s) = m \mid \forall e \in setMembers(s) \quad m(e) = value_{fe}(s, \langle e \rangle)$

To facilitate partial updates to sets, the **add/to**-rule and **remove/from**-rule are supported by the Set plugin (see Section 5.3.1). We have

$$\forall s \in \textsc{SetElement} \; isModifiableCollection(s)$$

The single addition of an element from a set, or the **add/to**-rule, results in an instruction to carry out a *setAddAction* action; the removal of a single element from a set, or the **remove/from**-rule, results in an instruction to perform a *setRemoveAction* action. For all $loc \in Location$ and $value \in Element$, we have

$$computeAddUpdate_{Set}(loc, value) \equiv \{\!\{\langle loc, value, setAddAction \rangle\}\!\}$$
$$computeRemoveUpdate_{Set}(loc, value) \equiv \{\!\{\langle loc, value, setRemoveAction \rangle\}\!\}$$

Notice that no checks are made to ensure that the value of the location is in fact a set. This is deferred to the aggregation phase.

Set Enumeration and Comprehension

The set plugin provides two methods of set description: namely set enumeration and set comprehension. With the former, one is able to explicitly describe the contents of a set by listing its individual elements:

Set Plugin: Set Enumeration

```
$( \{ {}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n \} ) \rightarrow$
    choose $i \in [1..n]$ with $\neg evaluated(\lambda_i)$
      $pos := \lambda_i$
    ifnone
      let $s = \{value(\lambda_i) \mid i \in [1..n]\}$ in
        $[\![pos]\!] := (undef, undef, setElement(s))$
```

The latter allows one to describe set contents algorithmically. There are many accepted syntactic and semantic variants; the Set plugin provides three variants which we believe encompass a wide range of algorithmically expressible finite sets. Given a set comprehension expression of the form

$$\{x_0 \textbf{ is } exp_0 \mid x_1 \textbf{ in } exp_1, \ldots, x_n \textbf{ in } exp_n \textbf{ with } exp_g\}$$

we refer to the free variable x_0 as the *specifier variable*, the expression exp_0 as the *specifier expression*, the free variables $x_1 \ldots x_n$ as the *constrainer variables*, $exp_1 \ldots exp_n$ as the *constrainer expression*, and exp_g as the *guard*.

The simplest variant of set comprehension binds the specifier variable to a constrainer expression producing a single enumerable element:

Set Plugin: Set Comprehension

```
$( \{ {}^{\alpha}x \mid {}^{\beta_1}x_1 \textbf{ in } {}^{\gamma_1}\boxed{?}_1 \} ) \rightarrow$
    if $x = x_1$ then
      if $\neg evaluated(\gamma_1)$ then
        $pos := \gamma_1$
      else
        if $enumerable(value(\gamma_1))$ then
          let $s = \{m \mid m \in enumerate(value(\gamma_1))\}$ in
            $[\![pos]\!] := (undef, undef, setElement(s))$
        else
          Error(`Free variables may only be bound to enumerable elements')
    else
      Error(`Constrainer variable must have same name as specifier variable')
```

Notice how we use the $setElement(s)$ mapping to get a Set element representation of the set s. This variant would support set comprehension expressions of the form $\{x \mid x \textbf{ in } X\}$ where X is an enumerable element.

A slightly more complex version supports set comprehensions of the form

$$\{x \mid x \textbf{ in } X, y_1 \textbf{ in } Y_1, \ldots, y_n \textbf{ in } Y_n \textbf{ with } \varphi\}$$

where X and Y_i's are enumerable elements and x and y_i's are free variables in φ. This form binds multiple constrainer variables to multiple constrainer expressions, and adds more fine grained control with a guard. The semantic definition of this form involves creating temporary logical variables for each constrainer variable and iterating their values over the values offered by their corresponding constrainer expressions and evaluating the guard for each combination of these values. A formal semantic definition is provided in Appendix A.5.4. This variant supports set comprehension expressions such as:

$$\{x \mid x \textbf{ in } X \textbf{ with } x > z\}$$
$$\{x \mid x \textbf{ in } \{1, 3, 5\},\ z \textbf{ in } \{2, 4, 6\} \textbf{ with } (x + z) \textbf{ in } \{3, 4, 5, 6, 7, 8, 9, 10\}\}$$

Finally the most complex variant of the form

$$\{x \textbf{ is } e \mid x_1 \textbf{ in } X_1, \ldots, x_n \textbf{ in } X_n \textbf{ with } \varphi\}$$

in which e is an expression, φ is a guard and x_1 to x_n are free variables in both e and φ, allows the specifier to be defined in terms of a specifier expression. In this form the constrainer variables are themselves expected to be present in the specifier expression, and this expression is re-evaluated for all possible combinations of the contrainer variables. Similar to the previous form, the semantics definition of this

```
$(\!| {}^{\alpha}\boxed{?} \cap {}^{\beta}\boxed{?} |\!)_{[675]}$   $\rightarrow$   choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                    $pos := \lambda$
                                 ifnone
                                    if $\forall x \in \{l, r\}$ $\textsc{SetElement}(x) \vee x = \mathsf{undef}_e$ then
                                       if $l = \mathsf{undef}_e \vee r = \mathsf{undef}_e$ then
                                          $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
                                       else
                                          let $v = \{x \mid x \in enumerate(l) \wedge x \in enumerate(r)\}$ in
                                             $[\![pos]\!] := (undef, undef, setElement(v))$
                                 where
                                    $l \equiv value(\alpha)$
                                    $r \equiv value(\beta)$
```

Notice that the evaluation of an operation results in a new Set element rather than modification of an existing Set element.

Aggregation Algorithm

The Set plugin is responsible for the aggregation of update instructions with $setAddAction$ and $setRemoveAction$ that add or remove elements to and from Set elements. The result of aggregation of set updates on a location will be a regular update assigning a new Set element (representing all the changes) to that location.

For every location with a set partial update, the Set plugin first checks the consistency of update instructions before performing the aggregation. The following requirements informally define the consistency of set update instructions [64]:

- If there is a regular update to a given location l along with partial updates:
  - All regular updates to l may only result in a Set element.
  - There cannot exist two regular updates to l resulting in two different values; this is a typical consistency requirement of regular updates.
  - The Set element S assigned by the regular update(s) on l must satisfy all the add and remove update instructions to l; i.e., $\forall\langle l, v_a, setAddAction\rangle \in updates,\ v_a \in S$ and $\forall\langle l, v_r, setRemoveAction\rangle \in updates,\ v_r \notin S$.
- If there are only partial updates to a given location l:
  - There cannot exist two update instructions adding and removing the same element e to location l.
  - The value of location l in the current state of the simulated machine must be a Set element.

The following rule defines the aggregation algorithm offered by the Set plugin; we have

$$aggregatorRule(setPlugin) \equiv @\mathsf{Aggregate}_{Set}$$

Set Plugin

```
**Aggregate**$_{Set}(uMset) \equiv$
  **local** $resultantUpdate$ **in**
  **seq**
    **result** := {}
  **next**
    **forall** $l \in locsToAggregate$ **do**
      **if** $regularUpdatesExist$ **then**
        **if** $inconsistentRegularUpdates \lor regularUpdateIsNotSet \lor addRemoveConflictWithRU$ **then**
          HandleInconsistentAggregation($l, uMset, setPlugin$)
         **else**
          **let** $resultantUpdate$ = GetRegularUpdate($l, uMset$) **in**
             **add** $resultantUpdate$ **to result**
      **else**
        **if** $addRemoveConflict \lor setNotInLocation$ **then**
          HandleInconsistentAggregation($l, uMset, setPlugin$)
        **else**
          **let** $resultantUpdate$ = BuildResultantUpdate($l, uMset$) **in**
            **add** $resultantUpdate$ **to result**
where
  $locsToAggregate \equiv \{l \mid \langle l, v, a\rangle \in uMset \land a \in \{setAddAction, setRemoveAction\}\}$
  $regularUpdatesExist \equiv \exists\langle l, v, updateAction\rangle \in uMset$
  $inconsistentRegularUpdates \equiv \exists\langle l, v_1, updateAction\rangle \in uMset,$
                                  $\exists\langle l, v_2, updateAction\rangle \in uMset, v_1 \neq v_2$
  $regularUpdateIsNotASet \equiv \exists\langle l, v, updateAction\rangle \in uMset, bkg(v) \neq$ "Set"
  $addRemoveConflictWithRU \equiv addConflictWithRU \lor removeConflictWithRU$
  $addConflictWithRU \equiv \exists\langle l, v_u, updateAction\rangle \in uMset,$
                          $\exists\langle l, v_a, setAddAction\rangle \in uMset,\ v_a \notin enumerate(v_u)$
  $removeConflictWithRU \equiv \exists\langle l, v_u, updateAction\rangle \in uMset,$
                          $\exists\langle l, v_r, setRemoveAction\rangle, \in uMset,\ v_r \in enumerate(v_u)$
  $addRemoveConflict \equiv \exists\langle l, v, setAddAction\rangle \in uMset, \exists\langle l, v, setRemoveAction\rangle \in uMset$
  $setNotInLocation \equiv bkg(getValue(l)) \neq$ "Set"
```

In the case where at least one regular update exists for a location, after checking the consistency of partial updates with the regular updates on that location, one of the regular updates will be chosen as the result of the aggregation.

Set Plugin

```
**GetRegularUpdate**($loc, uMset$) $\equiv$
  **choose** $u \in uMset$ **with** $uiLoc(u) = loc \land uiAction(u) = updateAction$ **do**
    **result** := $u$
  **forall** $u \in uMset$ **with** $uiLoc(u) = loc$ **do**
    $aggStatus(u, setPlugin)$ := $successful$
```

When there is no regular update for a location, all the partial updates are aggregated into a regular update assigning a new Set element to the location resulting from the addition and removal of elements from the value of the location in the current state.

Set Plugin

```
BuildResultantUpdate($l$, $uMset$) $\equiv$
  local $newSet$ [$newSet$ := $\{\}$] in
    seq
      forall $e \in enumerate(getValue(l))$ do
        if $\nexists\langle l, e, setRemoveAction\rangle \in uMset$ then
          add $e$ to $newSet$
      forall $\langle l, v, setAddAction\rangle \in uMset$ do
        add $v$ to $newSet$
    next
      result := $\langle l, setElement(newSet), updateAction\rangle$
      forall $u \in uMset$ with $uiLoc(u) = l$ do
        $aggStatus(u, setPlugin)$ := $successful$
```

Composition Algorithm

The Set plugin provides the semantics of sequential composition of Set partial updates. There are five cases to be considered:

1. If the location is not updated in the second step, all the updates of the first step are carried forward.

2. If the location is not updated in the first step, all the updates of the second step are carried forward.

3. If there is a regular update on the location in the second step (i.e., a Set element is assigned to the location in the second step), all the updates in the first step are discarded and the updates of the second step are carried forward.

4. If there is a regular update on the location in the first step and there are partial updates in the second step, the updates need to be aggregated into one regular update.

5. If there are only partial updates on the location in both the first and the second step, those partial updates in the first step that are overridden by the updates in the second step must be removed.

The Set composition algorithm, capturing the five cases above, is formally defined as follows:

Set Plugin

```
**Compose**$_{Set}(uMset_1, uMset_2) \equiv$
  seq
   result := $\{\!|\,|\!\}$
  next
   forall $l \in locsAffected$ do
    if $locHasAddRemove(uMset_1) \wedge \neg locUpdated(uMset_2)$ then
      forall $ui \in uMset_1$ with $uiLoc(ui) = l$ do
        add $ui$ to result
    else if $\neg locUpdated(uMset_1) \wedge locHasAddRemove(uMset_2)$ then
      forall $ui \in uMset_2$ with $uiLoc(ui) = l$ do
        add $ui$ to result
    else if $locHasAddRemove(uMset_2) \wedge locRegularUpdate(uMset_2)$ then
      forall $ui \in uMset_2$ with $uiLoc(ui) = l$ do
        add $ui$ to result
    else if $locHasAddRemove(uMset_2) \wedge locRegularUpdate(uMset_1)$ then
      add SetAggregateLocation($l, uMset_1, uMset_2$) to result
    else if $locHasAddRemove(uMset_1) \wedge locHasAddRemove(uMset_2)$ then
      forall $ui \in$ EradicateConflictingUpdates($l, uMset_1, uMset_2$) do
        add $ui$ to result

where
  $locsAffected \equiv \{l_1 \mid \langle l_1, v, a\rangle \in uMset_1\} \cup \{l_2 \mid \langle l_2, v, a\rangle \in uMset_2\}$
  $locHasAddRemove(uMset) \equiv \exists\langle l, v, a\rangle \in uMset, a \in \{setAddAction, setRemoveAction\}$
  $locRegularUpdate(uMset) \equiv \exists\langle l, v, a\rangle \in uMset, a = updateAction$
  $locUpdated(uMset) \equiv \exists\langle l, v, a\rangle \in uMset$
```

In case (4), the regular update produced is created by aggregating the partial updates in the second step, assuming that the location currently contains the value of the regular update from the first step. The following rule formally defines the semantics of this aggregation.

Set Plugin

```
**SetAggregateLocation**$(loc, uMset_1, uMset_2) \equiv$
  return $resultantUpdate$ in
    local $newSet$ [$newSet$ := {}] in
        seq
          forall $e \in enumerate(getLocRegularUpdateValue(uMset_1))$
            if $\nexists\langle loc, e, setRemoveAction\rangle \in uMset_2$ do
              add $e$ to $newSet$
           forall $\langle loc, v, setAddAction\rangle \in uMset_2$ do
             add $v$ to $newSet$
        next
          $resultantUpdate := \langle loc, setElement(newSet), updateAction\rangle$
where
  $getLocRegularUpdateValue(uMset) \equiv\ v\ s.t.\ \langle loc, v, a\rangle \in uMset \wedge a = updateAction$
```

Partial update instructions occurring in a sequence may nullify one another. In case (5), we remove the updates that fall into one of these categories:

- For any location, addition of an element e in the first step followed by the removal of the same element e in the second step, clearly causes no change to the resulting Set element. Update instructions containing both these opposing actions on the same location are removed from the composed update multiset.
- For any location, removal of an element e in the first step is neutralized by the addition of the same element e in the second step. Thus, such removal update instructions should be excluded from the composed update multiset.

The following rule formally defines the composition behavior in case (5):

```
                                                                        Set Plugin
EradicateConflictingSetUpdates($loc, uMset_1, uMset_2$) $\equiv$
  return $remainingUpdates$ in
    seq
      $remainingUpdates := \{\!\!\{\}\!\!\}$
    next
      forall $v \in locValues$ do
        if $locValAct(uMset_1, v, setAddAction) \wedge locValAct(uMset_2, v, setRemoveAction)$ then
          skip
        else if $locValAct(uMset_1, v, setRemoveAction) \wedge locValAct(uMset_2, v, setAddAction)$ then
          forall $ui \in \{\!\!\{\langle loc, v, setAddAction\rangle \in uMset_2\}\!\!\}$ do
            add $ui$ to $remainingUpdates$
        else
          forall $ui \in getAllLocValUpdates$ do
            add $ui$ to $remainingUpdates$
where
  $locValues \equiv \{v_1 \mid \langle loc, v_1, a_1\rangle \in uMset_1\} \cup \{v_2 \mid \langle loc, v_2, a_2\rangle \in uMset_2\}$
  $locValAct(uMset, v, a) \equiv \exists\langle loc, v, a\rangle \in uMset$
  $getAllLocValUpdates \equiv \{\langle loc, v, a_1\rangle \in uMset_1\} \cup \{\langle loc, v, a_2\rangle \in uMset_2\}$
```

5.3.3 The Bag Plugin

The Bag plugin extends the CoreASM language with the background of finite *Bags* or multisets. The background of Bag elements (or Multiset elements) is defined by $bagBack \in$ BACKGROUNDELEMENT; we have

$$\begin{aligned} name(bagBack) &= \text{“BAG”} \\ newValue(bagBack) &= emptyBag \end{aligned}$$

We model Bag elements as values of a domain BAGELEMENT. The following functions define the interface of Bag elements and provide a mapping between Bag elements and the multisets of elements they represent:

- $\mathit{bagElement}: \textsc{Multiset}(\textsc{Element}) \mapsto \textsc{BagElement}$
  for every multiset of elements, returns a bag element representation of that multiset.

- $\mathit{bagElement}^f : (\textsc{Element} \mapsto \mathbb{N}) \mapsto \textsc{BagElement}$
  for every mapping of elements to positive integers (multiplicity function), returns a bag element with the given multiplicity function.

- $\mathit{bagValue}: \textsc{BagElement} \mapsto \textsc{Multiset}(\textsc{Element})$
  for every bag element, returns the multiset of elements that the bag represents.

- $\mathit{bagMultiplicity}: \textsc{BagElement} \mapsto (\textsc{Element} \mapsto \mathbb{N})$
  for every bag element, returns the multiplicity function of the multiset it represents. The value of this function is zero for all the elements that are not in the bag.

- $\mathit{bagDomain}: \textsc{BagElement} \mapsto \textsc{Set}(\textsc{Element})$
  for every bag element, returns the set of all the elements that are in the bag.

For all $b \in \textsc{BagElement}$ we have:

- $\mathit{bkg}(b)$:= "Bag".

- $\forall b' \in \textsc{BagElement} \quad \mathit{equal}_{\mathit{Bag}}(b, b') \equiv$
  $\mathit{bagDomain}(b) = \mathit{bagDomain}(b')$
  $\wedge \forall e \in \mathit{bagDomain}(b) \quad \mathit{bagMultiplicity}(b)(e) = \mathit{bagMultiplicity}(b')(e)$

- $\mathit{enumerable}(b)$
  All bag elements are enumerable.

- $\mathit{enumerate}_{\mathit{Bag}}(b) = \mathit{bagValue}(b)$.

- $b \in \textsc{FunctionElement}$
  All bag elements also behave as functions.

- $\mathit{class}_{fe}(b) = \mathit{static}$

- $\forall e \in \textsc{Element} \quad \mathit{value}_{fe}(b, \langle e \rangle) \equiv \mathit{numberElement}(\mathit{bagMultiplicity}(b)(e))$

- $b \in \textsc{AbstractMapElement}$
  All bag elements are abstract map elements.

- $\mathit{getMap}_{\mathit{Bag}}(b) = m \mid \forall e \in \mathit{bagDomain}(b) \quad m(e) = \mathit{value}_{fe}(b, \langle e \rangle)$

To facilitate partial updates of Bag elements, the **add/to**-rule and **remove/from**-rule are supported by the Bag plugin (see Section 5.3.1). We have

$$\forall b \in \textsc{BagElement} \ \ \mathit{isModifiableCollection}(b)$$

Since incremental updates on bags do not come with much constraints as for sets (due to multiplicity of elements), instead of using different update actions for adding/removing elements to/from bags, Bag plugin uses a more general action, $bagUpdateAction$, with special values (elements) that also include the actions of adding, removing, or an ordered combination of adding or removing of elements; the latter is useful in composing incremental updates on bags:

$$
\begin{aligned}
&computeAddUpdate_{Bag}(loc, value) \equiv\\
&\qquad \{\!|\langle loc, bagUpdateElement(\text{“add”}, value), bagUpdateAction\rangle|\!\}\\
&computeRemoveUpdate_{Bag}(loc, value) \equiv\\
&\qquad \{\!|\langle loc, bagUpdateElement(\text{“remove”}, value), bagUpdateAction\rangle|\!\}
\end{aligned}
$$

Expression Forms

The interpreter is extended with the following Bag enumeration forms:

```
                                                        Bag Plugin
$\llparenthesis << \quad >> \rrparenthesis$   →   $[\![pos]\!]$ := $(undef, undef, emptyBag)$

$\llparenthesis << {}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n \; >> \rrparenthesis$   →   choose $i \in [1..n]$ with $\neg evaluated(\lambda_i)$
                                                  $pos := \lambda_i$
                                                ifnone
                                                  let $m = \{\!|value(\lambda_i) \mid i \in [1..n]|\!\}$ in
                                                    $[\![pos]\!] := (undef, undef, bagElement(m))$
```

Various forms of bag comprehension similar in syntax and semantics to those of sets (see Section 5.3.2) is also introduced by the Bag plugin.

Operators

Bag plugin provides the following four operators on Bag elements: ∩ (multiset intersection), \ (multiset difference), ∪ (multiset union), and + (multiset join) as defined below:

```
                                                        Bag Plugin
$\llparenthesis {}^{\alpha}\boxed{?} \cap {}^{\beta}\boxed{?} \rrparenthesis_{[675]}$   →   choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
      $pos := \lambda$
    ifnone
      let $l = value(\alpha), r = value(\beta)$ in
        if $\textsc{BagElement}(l) \wedge \textsc{BagElement}(r)$ then
          let $f = \{x \mapsto y \mid x = (bagDomain(l) \cap bagDomain(r))$
                      $\wedge\, y = min(bagValue(l)(x), bagValue(r)(x))\}$ in
            $[\![pos]\!] := (undef, undef, bagElement^{f}(f))$
        else
          if $\forall x \in \{l, r\}\ \textsc{BagElement}(x) \vee x = \mathsf{undef}_e$ then
            $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

```
$(\!|\, {}^{\alpha}\boxed{?} \backslash {}^{\beta}\boxed{?} \,|\!)_{[650]} \rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                        $pos := \lambda$
                      ifnone
                        let $l = value(\alpha), r = value(\beta)$ in
                          if $\textsc{BagElement}(l) \wedge \textsc{BagElement}(r)$ then
                            let $f = \{x \mapsto y \mid x \in bagDomain(l) \cup bagDomain(r)$
                                        $\wedge\, y = max(0, bagValue(l)(x) - bagValue(r)(x))\}$ in
                              $[\![pos]\!] := (undef, undef, bagElement^{f}(f))$
                          else
                            if $\forall x \in \{l, r\}\ \textsc{BagElement}(x) \vee x = \mathsf{undef}_e$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$

$(\!|\, {}^{\alpha}\boxed{?} \cup {}^{\beta}\boxed{?} \,|\!)_{[650]} \rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                        $pos := \lambda$
                      ifnone
                        let $l = value(\alpha), r = value(\beta)$ in
                          if $\textsc{BagElement}(l) \wedge \textsc{BagElement}(r)$ then
                            let $f = \{x \mapsto y \mid x \in bagDomain(l) \cup bagDomain(r)$
                                        $\wedge\, y = max(bagValue(l)(x), bagValue(r)(x))\}$ in
                              $[\![pos]\!] := (undef, undef, bagElement^{f}(f))$
                          else
                            if $\forall x \in \{l, r\}\ \textsc{BagElement}(x) \vee x = \mathsf{undef}_e$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$

$(\!|\, {}^{\alpha}\boxed{?} + {}^{\beta}\boxed{?} \,|\!)_{[750]} \rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                        $pos := \lambda$
                      ifnone
                        let $l = value(\alpha), r = value(\beta)$ in
                          if $\textsc{BagElement}(l) \wedge \textsc{BagElement}(r)$ then
                            let $f = \{x \mapsto y \mid x \in bagDomain(l) \cup bagDomain(r)$
                                        $\wedge\, y = bagValue(l)(x) + bagValue(r)(x)\}$ in
                              $[\![pos]\!] := (undef, undef, bagElement^{f}(f))$
                          else
                            if $\forall x \in \{l, r\}\ \textsc{BagElement}(x) \vee x = \mathsf{undef}_e$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

5.3.4 The List Plugin

The List plugin extends the CoreASM language providing the background of lists (sequence of elements) with corresponding operators and rule forms. We denote the background of List elements by $listBkg \in \textsc{BackgroundElement}$; we have

$$name(listBkg) = \text{“LIST”}$$
$$newValue(listBkg) = emptyList$$

List elements are values of the domain $\textsc{ListElement}$. The following functions define the interface of list elements and provide a mapping between List elements and the sequence of elements they represent.

- $\mathit{listElement} : \textsc{List}(\textsc{Element}) \mapsto \textsc{ListElement}$
  returns a list element representing the given sequence of elements.

- $\mathit{listValue} : \textsc{ListElement} \mapsto \textsc{List}(\textsc{Element})$
  returns the sequence of elements that are represented by the given list element,

- $\mathit{head}_{le} : \textsc{ListElement} \mapsto \textsc{Element}$
  $\mathit{last}_{le} : \textsc{ListElement} \mapsto \textsc{Element}$
  return the first and last elements of the list, or undef_e if the list is empty.

- $\mathit{tail}_{le} : \textsc{ListElement} \mapsto \textsc{ListElement}$
  returns the tail of the list excluding its first element, or an empty list if the list has only one element.

- $\mathit{cons}_{le} : \textsc{Element} \times \textsc{ListElement} \mapsto \textsc{ListElement}$
  $\mathit{cons}_{le}(e, l)$ constructs a new list with e as its head and l as its tail.

- $\mathit{concat}_{le} : \textsc{ListElement} \times \textsc{ListElement} \mapsto \textsc{ListElement}$
  $\mathit{concat}_{le}(l_1, l_2) \equiv \mathit{cons}_{le}(\mathit{head}_{le}(l_1), \mathit{concat}_{le}(\mathit{tail}_{le}(l_1), l_2))$

- $\mathit{listItem}_{le} : \textsc{ListElement} \times \mathbb{N} \mapsto \textsc{Element}$
  $\mathit{listItem}_{le}(l, i) \equiv \mathit{listValue}(l)(i)$

- $\mathit{take}_{le} : \textsc{ListElement} \times \mathbb{N} \mapsto \textsc{ListElement}$
  $\mathit{take}_{le}(\mathit{list}, i)$ returns a list element containing the first i elements of list as a list element. The first element of the list is at index 1.

- $\mathit{drop}_{le} : \textsc{ListElement} \times \mathbb{N} \mapsto \textsc{ListElement}$
  $\mathit{drop}_{le}(\mathit{list}, i)$ returns a list element containing what is left after dropping the first i elements of the list list. The first element of the list is at index 1.

For every $l \in \textsc{ListElement}$, we have

- $\mathit{bkg}(l)$ = "List"

- $\forall l' \in \textsc{ListElement} \quad \mathit{equal}_{\mathit{List}}(l, l') \equiv \mathit{listValue}(l) = \mathit{listValue}(l')$

- $\mathit{enumerable}(l)$
  All list elements are enumerable.

- $\mathit{enumerate}_{\mathit{List}}(l) = \mathit{listValue}(l)$.

- $l \in \textsc{FunctionElement}$
  All list elements also behave as functions.

- $\mathit{class}_{fe}(l) = \mathit{static}$

- $\forall ne \in \textsc{NumberElement} \quad \mathit{value}_{fe}(l, \langle ne \rangle) \equiv$
  $$\begin{cases} \mathit{listItem}_{le}(l, \mathit{numericValue}(ne)), & \text{if } \mathit{listItem}_{le}(l, \mathit{numericValue}(ne)) \neq \mathit{undef}; \\ \mathsf{undef}_e, & \text{otherwise.} \end{cases}$$

- $l \in$ AbstractMapElement
  All List elements are abstract map elements.
- $getMap_{List}(l) = m \mid \forall e \in [1..|enumerate_{List}(l)|]\;\; m(e) = value_{fe}(l, \langle e \rangle)$

Every list element is considered to be a modifiable collection, so we have

$$\forall l \in \textsc{ListElement}\;\; isModifiableCollection(l)$$

However, List plugin does not offer partial updates on List elements; hence, adding and removing elements to and from List elements cannot be done incrementally. As a result, $computeAddUpdate_{List}$ and $computeRemoveUpdate_{List}$ on lists return an update instruction with a regular update action defined as:

$$\begin{array}{l} computeAddUpdate_{List}(loc, value) \equiv \\ \quad \{\!|\langle loc, concat_{le}(getValue(loc), listElement(\langle value \rangle)), updateAction\rangle|\!\} \\ computeRemoveUpdate_{List}(loc, value) \equiv \\ \quad \begin{cases} \{\!|\langle loc, concat_{le}(left, right), updateAction\rangle|\!\}, & \text{if } |indices(getValue(loc))| > 0; \\ \{\!|\,|\!\}, & \text{otherwise.} \end{cases} \end{array}$$

where

$$\begin{array}{l} indices(le) = \{j \mid j \in [1..|listValue(le)|] \wedge listValue(le)(j) = value\} \\ left = take_{le}(getValue(loc), m - 1) \\ right = drop_{le}(getValue(loc), m) \\ m = min(indices(getValue(loc))) \end{array}$$

Expression Forms

The List plugin extends the interpreter to support List comprehension:

```
                                                                List Plugin
$(\!|\,[\;]\,|\!)$                          $\rightarrow$   let $newList = newValue(listBkg)$ in
                                                    $[\![pos]\!] := (undef, undef, newList)$

$(\!|\,[\;{}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n\;]\,|\!)$   $\rightarrow$   choose $i \in [1..n]$ with $\neg evaluated(\lambda_i)$
                                                    $pos := \lambda_i$
                                                  ifnone
                                                    let $l = \langle value(\lambda_1), \ldots, value(\lambda_n)\rangle$ in
                                                      $[\![pos]\!] := (undef, undef, listElement(l))$
```

To facilitate locating a specific element in a List element, the List plugin also offers the following expression form that searches a List element for the occurrence of an element and returns an index to the element of interest. If there is no such element in the list, the result will be undef_e. If the element appears more than once in the list, one index will be returned non-deterministically.

List Plugin : Search

```
$(\!|$ indexof $^{\alpha}\boxed{e}$ in $^{\beta}\boxed{e}$ $|\!)$ $\rightarrow$  choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
                              $pos := \tau$
                            ifnone
                              let $e = value(\alpha), v = value(\beta)$ in
                                if $v \in$ ListElement then
                                  let $l = listValue(v)$ in
                                    choose $i \in [1..|l|]$ with $l(i) = e$ do
                                      $[\![pos]\!] := (undef, undef, numberElement(i))$
                                    ifnone
                                      $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

In addition, the following expression forms, return an index to the first and the last occurrence of an element in a list.

List Plugin : Search

```
$(\!|$ first indexof $^{\alpha}\boxed{e}$ in $^{\beta}\boxed{e}$ $|\!)$ $\rightarrow$
      choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
        $pos := \tau$
      ifnone
        let $e = value(\alpha), v = value(\beta)$ in
          if $v \in$ ListElement then
            let $l = listValue(v)$ in
              let $indices = \{j \mid j \in [1..|l|] \wedge l(j) = e\}$ in
                if $|indices| > 0$ then
                  $[\![pos]\!] := (undef, undef, numberElement(min(indices)))$
                else
                  $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$

$(\!|$ last indexof $^{\alpha}\boxed{e}$ in $^{\beta}\boxed{e}$ $|\!)$ $\rightarrow$
      // Similar to above; replace $min(indices)$ by $max(indices)$.
```

Operators

The List plugin provides the following concatenation operator on List elements:

List Plugin : Concatenation

```
$(\!|\,^{\alpha}\boxed{?} + {}^{\beta}\boxed{?}\,|\!)_{[750]}$ $\rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                         $pos := \lambda$
                       ifnone
                         let $l = value(\alpha), r = value(\beta)$ in
                           if $l \in$ ListElement $\wedge\ r \in$ ListElement then
                             $[\![pos]\!] := (undef, undef, concat_{le}(l, r))$
                           else
                             if $\forall x \in \{l, r\}\ x \in$ ListElement $\vee\ x = \mathsf{undef}_e$ then
                               $[\![pos]\!] := (undef, undef, \mathsf{undef}_e)$
```

Rule Forms

The List plugin extends the interpreter of the engine to provide the following rule forms facilitating *shifting* of List elements one index to the left or right. In shift left, the first element of the list is dropped into the given location. In shift right, the last element of the list is dropped into the given location.

List Plugin

```
$\llparenthesis$ shift left$^\alpha\boxed{\phantom{x}}$ into$^\beta\boxed{l}$ $\rrparenthesis \rightarrow$
        choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
          $pos$ := $\tau$
        ifnone
          if $value(\alpha) \in$ LISTELEMENT then
            if $loc(\beta) \neq undef$ then
              let $updates$ = $\{\!|\langle loc(\beta), head_{le}(value(\alpha)), updateAction\rangle,$
                                $\langle loc(\alpha), tail_{le}(value(\alpha)), updateAction\rangle|\!\}$
                $[\![pos]\!]$ := $(undef, updates, undef)$
            else
              Error(`Cannot shift list to a non-location.')

$\llparenthesis$ shift right$^\alpha\boxed{\phantom{x}}$ into$^\beta\boxed{l}$ $\rrparenthesis \rightarrow$
        choose $\tau \in \{\alpha, \beta\}$ with $\neg evaluated(\tau)$
          $pos$ := $\tau$
        ifnone
          if $value(\alpha) \in$ LISTELEMENT then
            if $loc(\beta) \neq undef$ then
              let $le$ = $value(\alpha), l$ = $listValue(le)$ in
                if $|l| \leq 1$ then
                  let $updates$ = $\{\!|\langle loc(\beta), last_{le}(le), updateAction\rangle,$
                                    $\langle loc(\alpha), emptyList, updateAction\rangle|\!\}$
                    $[\![pos]\!]$ := $(undef, updates, undef)$
                else
                  let $updates$ = $\{\!|\langle loc(\beta), last_{le}(le), updateAction\rangle,$
                                    $\langle loc(\alpha), take_{le}(le, |l| - 1), updateAction\rangle|\!\}$
                    $[\![pos]\!]$ := $(undef, updates, undef)$
            else
              Error(`Cannot shift list to a non-location.')
```

Functions

The List plugin also extends the vocabulary of the engine to provide the following functions defined on List elements:

- `head: LIST -> ELEMENT`
  $value_{fe}(headFunction, \langle l\rangle) = head_{le}(l)$
- `last: LIST -> ELEMENT`
  $value_{fe}(lastFunction, \langle l\rangle) = last_{le}(l)$

- `tail: LIST -> LIST`
  $value_{fe}(tailFunction, \langle l \rangle) = tail_{le}(l)$
- `cons: ELEMENT * LIST -> LIST`
  $value_{fe}(consFunction, \langle e, l \rangle) = cons_{le}(e, l)$
- `nth: LIST * NUMBER -> ELEMENT`
  $value_{fe}(nthFunction, \langle l, i \rangle) = listItem_{le}(l, numricValue(i))$
- `take: LIST * NUMBER -> LIST`
  $value_{fe}(takeFunction, \langle l, i \rangle) = take_{le}(l, numricValue(i))$
- `drop: LIST * NUMBER -> LIST`
  $value_{fe}(dropFunction, \langle l, i \rangle) = drop_{le}(l, numricValue(i))$
- `reverse: LIST -> LIST`
  $value_{fe}(reverseFunction, \langle l \rangle) =$
  $$\begin{cases} emptyList, & \text{if } |listValue(l)| = 0; \\ reverse(l), & \text{otherwise.} \end{cases}$$
  where
  $reverse(l) \equiv l'$ s.t. $\forall_{i \in [1..|listValue(l)|]}\ listItem_{le}(l', i) = listItem_{le}(l, |listValue(l)| - i + 1)$
- `indexes: LIST -> LIST`
  $value_{fe}(indexesFunction, \langle l \rangle) = listElement(\langle 1, \ldots, |listValue(l)| \rangle)$
- `indices: LIST -> LIST`
  same as `indexes`.
- `setnth: LIST * NUMBER * ELEMENT -> LIST`
  $value_{fe}(setnthFunction, \langle l, n, e \rangle) =$
  $$\begin{cases} l' \text{ s.t. } listItem_{le}(l', numericValue(n)) = e, & \text{if } 1 \leq n \leq |listValue(l)|; \\ \text{undef}_e, & \text{otherwise.} \end{cases}$$

5.3.5 The Queue Plugin

The Queue plugin does not provide any new type domain but it provides two rule forms that operate on Lists elements as queues: **enqueue** and **dequeue**. The former adds an element to end of the list, and the latter removes an element from the head of the list. We present here a formal definition of these two rule forms:

Queue Plugin

```
$\llparenthesis \textbf{enqueue}^{\alpha}\boxed{e}\ \textbf{into}^{\beta}\square \rrparenthesis \rightarrow$   $pos := \beta$

$\llparenthesis \textbf{enqueue}^{\alpha}\boxed{e}\ \textbf{into}^{\beta} l \rrparenthesis \rightarrow$   $\textbf{if } value(\beta) \in \textsc{ListElement} \textbf{ then}$
    $pos := \alpha$
$\textbf{else}$
    Error(`Cannot enqueue into a non-list.')

$\llparenthesis \textbf{enqueue}^{\alpha} v\ \textbf{into}^{\beta} l \rrparenthesis \rightarrow$   $\textbf{let } newList = concat_{le}(value(\beta), listElement(\langle v \rangle)) \textbf{ in}$
    $[\![pos]\!] := (undef, \{\!\!\{\langle l, newList, updateAction\rangle\}\!\!\}, undef)$

$\llparenthesis \textbf{dequeue}^{\alpha}\boxed{l}\ \textbf{from}^{\beta}\boxed{l}\ \rrparenthesis \rightarrow$   $pos := \beta$

$\llparenthesis \textbf{dequeue}^{\alpha}\boxed{l}\ \textbf{from}^{\beta} l_2 \rrparenthesis \rightarrow$   $\textbf{if } value(\beta) \in \textsc{ListElement} \textbf{ then}$
    $\textbf{if } |listValue(value(\beta))| > 0 \textbf{ then}$
        $pos := \alpha$
    $\textbf{else}$
        Error(`Cannot dequeue from an empty queue.')
$\textbf{else}$
    Error(`Cannot dequeue into a non-list.')

$\llparenthesis \textbf{dequeue}^{\alpha} l_1\ \textbf{from}^{\beta} l_2 \rrparenthesis \rightarrow$   $\textbf{let } u_1 = \langle l_1, head_{le}(value(\beta)), updateAction\rangle,$
        $u_2 = \langle l_2, tail_{le}(value(\beta)), updateAction\rangle \textbf{ in}$
    $[\![pos]\!] := (undef, \{\!\!\{u_1, u_2\}\!\!\}, undef)$
```

5.3.6 The Stack Plugin

Similar to the Queue plugin introduced above, the Stack plugin also does not provide any new type domain but it provides two rule forms that operate on Lists as stacks: **push** and **pop**. The former one, pushes an element at the head of a list and the latter one removes the first element of the list.

Stack Plugin

```
$\llparenthesis \textbf{push}^{\alpha}\boxed{e}\ \textbf{into}^{\beta}\square \rrparenthesis \rightarrow$   $pos := \beta$

$\llparenthesis \textbf{push}^{\alpha}\boxed{e}\ \textbf{into}^{\beta} l \rrparenthesis \rightarrow$   $\textbf{if } value(\beta) \in \textsc{ListElement} \textbf{ then}$
    $pos := \alpha$
$\textbf{else}$
    Error(`Cannot push into a non-list.')

$\llparenthesis \textbf{push}^{\alpha} v\ \textbf{into}^{\beta} l \rrparenthesis \rightarrow$   $\textbf{let } newList = cons_{le}(v, value(\beta)) \textbf{ in}$
    $[\![pos]\!] := (undef, \{\!\!\{\langle l, newList, updateAction\rangle\}\!\!\}, undef)$
```

```
$(\!| \mathbf{pop}^{\alpha}\boxed{l}\ \ \mathbf{from}^{\beta}\boxed{l}\ |\!)$    $\rightarrow$    $pos := \beta$

$(\!| \mathbf{pop}^{\alpha}\boxed{l}\ \ \mathbf{from}^{\beta} l_2 |\!)$    $\rightarrow$    $\mathbf{if}\ value(\beta) \in \textsc{ListElement}\ \mathbf{then}$
                                  $\mathbf{if}\ |listValue(value(\beta))| > 0\ \mathbf{then}$
                                    $pos := \alpha$
                                  $\mathbf{else}$
                                    Error(`Cannot pop from an empty stack.')
                                $\mathbf{else}$
                                  Error(`Cannot pop from a non-list.')

$(\!| \mathbf{pop}^{\alpha} l_1\ \mathbf{from}^{\beta} l_2 |\!)$    $\rightarrow$    $\mathbf{let}\ u_1 = \langle l_1, head_{le}(v), updateAction\rangle,$
                                    $u_2 = \langle l_2, tail_{le}(v), updateAction\rangle\ \mathbf{in}$
                                  $[\![pos]\!] := (undef, \{\!|u_1, u_2|\!\}, undef)$
```

5.3.7 The Map Plugin

The Map plugin extends CoreASM by providing the background of Map elements and the corresponding operators and rule forms defined on them. The background of map elements is denoted by $mapBkg \in \textsc{BackgroundElement}$; we have

$$\begin{aligned} name(mapBkg) &= \text{“MAP”} \\ newValue(mapBkg) &= emptyMap \end{aligned}$$

Map elements are values of the domain $\textsc{MapElement}$. The following functions define the interface of map elements and provide a mapping between Map elements to the unary functions or sets of pairs they represent:

- $mapElement : (\textsc{Element} \mapsto \textsc{Element}) \mapsto \textsc{MapElement}$
  returns a map element representing the given mapping of elements to elements.
- $mapElementFromPairs : \textsc{Set}(\textsc{ListElement}) \mapsto \textsc{MapElement}$
  if the given set consists of pairs of elements (lists of size two) of the form $[k_i, v_i]$ such that $\forall [k_i, v_i]\ \nexists [k_j, v_j]\ \ k_i = k_j \wedge v_i \neq v_j$, this function returns a map element representing a mapping of k_is to v_is; otherwise, returns undef_e.
- $mapValue : \textsc{MapElement} \mapsto (\textsc{Element} \mapsto \textsc{Element})$
  returns the mapping (from elements to elements) represented by the given map element.
- $keyset : \textsc{MapElement} \mapsto \textsc{Set}(\textsc{Element})$
  $\forall m \in \textsc{MapElement}, keyset(m) \equiv domain(mapValue(m))$
- $valueset : \textsc{MapElement} \mapsto \textsc{Set}(\textsc{Element})$
  $\forall m \in \textsc{MapElement}, valueset(m) \equiv range(mapValue(m))$

For every $m \in \textsc{MapElement}$, we have

- $\mathit{bkg}(m) = \text{“Map”}$
- $\forall m' \in \textsc{MapElement} \;\; \mathit{equal}_{Map}(m, m') \equiv$
  $\mathit{keyset}(m) = \mathit{keyset}(m') \wedge \forall e \in \mathit{keyset}(m) \;\; \mathit{mapValue}(m')(e) = \mathit{mapValue}(m)(e)$
- $\mathit{enumerable}(m)$
  All map elements are enumerable.
- $\mathit{enumerate}_{Map}(m) = \{\mathit{listElement}(\langle k, v\rangle) \mid k \in \mathit{keyset}(m) \wedge v = \mathit{mapValue}(m)(k)\}$.
- $m \in \textsc{FunctionElement}$
  All map elements also behave as functions.
- $\mathit{class}_{fe}(m) = \mathit{static}$
- $\forall e \in \textsc{Element} \;\; \mathit{value}_{fe}(m, \langle e\rangle) \equiv$
  $$\begin{cases} \mathit{mapValue}(m)(e), & \text{if } \mathit{mapValue}(m)(e) \neq \mathit{undef}; \\ \mathsf{undef}_e, & \text{otherwise.} \end{cases}$$
- $m \in \textsc{AbstractMapElement}$
  All map elements are abstract map elements.
- $\mathit{getMap}_{Map}(m) = \mathit{mapValue}(m)$

Every map element is considered to be a modifiable collection, so we have

$$\forall m \in \textsc{MapElement} \;\; \mathit{isModifiableCollection}(m)$$

However, Map plugin does not offer partial updates on Map elements; hence, adding and removing elements to and from Map elements cannot be done incrementally. As a result, $\mathit{computeAddUpdate}_{Map}$ and $\mathit{computeRemoveUpdate}_{Map}$ on maps return an update instruction with a regular update action defined as:

$$\mathit{computeAddUpdate}_{Map}(loc, value) \equiv$$
$$\begin{cases} \lbrace\!\vert \langle loc, \mathit{map}_{loc} \oplus \mathit{map}_{val}, \mathit{updateAction}\rangle \vert\!\rbrace & \text{if } \textsc{AbstractMapElement}(value); \\ \mathsf{undef}_e, & \text{otherwise.} \end{cases}$$
$$\mathit{computeRemoveUpdate}_{List}(loc, value) \equiv$$
$$\begin{cases} \lbrace\!\vert \langle loc, \mathit{map}_{loc} \ominus \mathit{map}_{val}, \mathit{updateAction}\rangle \vert\!\rbrace & \text{if } \textsc{MapElement}(value); \\ \lbrace\!\vert \langle loc, \mathit{map}_{loc} \otimes \mathit{enumerate}(value), \mathit{upAdateAction}\rangle \vert\!\rbrace, & \text{if } \neg\textsc{MapElement}(value) \\ & \quad \wedge\ \mathit{enumerable}(value); \\ \lbrace\!\vert \langle loc, \mathit{map}_{loc} \otimes \{value\}, \mathit{updateAction}\rangle \vert\!\rbrace, & \text{otherwise.} \end{cases}$$

where[4]

$$\begin{aligned}
map_{loc} &= mapElement(mapValue(getValue(loc))\\
map_{val} &= getMap_{bkg(value)}(value)\\
m_1 \oplus m_2 &= m_3 \mid \forall e \in \textsc{Element} \;\; m_3(e) = \begin{cases} m_2(e), & \text{if } m_2(e) \neq undef;\\ m_1(e), & \text{otherwise.}\end{cases}\\
m_1 \ominus m_2 &= m_3 \mid \forall e \in \textsc{Element} \;\; m_3(e) = \begin{cases} m_1(e), & \text{if } m_1(e) \neq m_2(e);\\ undef, & \text{otherwise.}\end{cases}\\
m \otimes s &= m' \mid \forall e \in \textsc{Element} \;\; m'(e) = \begin{cases} m(e), & \text{if } e \notin s;\\ undef, & \text{otherwise.}\end{cases}
\end{aligned}$$

Expression Forms

The Map plugin extends the interpreter of the CoreASM engine with the following map comprehension forms:

Map Plugin

```
$(\!|\, \{\texttt{->}\} \,|\!) \rightarrow \;\; [\![pos]\!] := (undef, undef, emptyMap)$

$(\!|\, \{^{\lambda_1}\boxed{?}\texttt{->}\; ^{\lambda_2}\boxed{?},\; \ldots,\; ^{\lambda_{2n-1}}\boxed{?}\texttt{->}\; ^{\lambda_{2n}}\boxed{?}\} \,|\!) \rightarrow$
        $\textbf{choose } i \in [1..2n] \textbf{ with } \neg evaluated(\lambda_i)$
          $pos := \lambda_i$
        $\textbf{ifnone}$
          $\textbf{let } pairs = \{listElement(\langle \lambda_{2i-1}, \lambda_{2i} \rangle) \mid i \in [1..n]\} \textbf{ in}$
            $[\![pos]\!] := (undef, undef, mapElementFromPairs(pairs))$
```

Functions

The vocabulary of the CoreASM engine is also extended with the following two functions mapping Map elements to sets of pairs and vice versa:

- `toMap: ELEMENT -> MAP`
  $value_{fe}(toMapFunction, \langle e \rangle) =$
  $$\begin{cases} mapElementFromPairs(\{x \mid x \in enumerate(e)\}), & \text{if } enumerable(e);\\ \text{undef}_e, & \text{otherwise.}\end{cases}$$

- `mapToPairs: MAP -> SET`
  $value_{fe}(mapToPairsFunction, \langle m \rangle) =$
  $$\begin{cases} setElement(enumerate(m)), & \text{if } m \in \textsc{MapElement};\\ \text{undef}_e, & \text{otherwise.}\end{cases}$$

[4] Here, the definitions of $\oplus$, $\ominus$, and $\otimes$ are local for these formula and should not be mistaken by other definitions throughout this document.

5.4 Auxiliary Plugins

In addition to the plugins addressed so far, CoreASM comes with a number of auxiliary plugins that extend the kernel of CoreASM with concepts, constructs and functionalities that are particularly useful in execution and analysis of specifications. Here, we present those auxiliary plugins that are available as part of the current edition of CoreASM.

5.4.1 The Signature Plugin

The CoreASM language is in principle an untyped language.[5] While a typeless language is desirable for writing initial specifications, defining the types of values and the signatures of functions used in more concrete specifications often add useful semantic information. Such information not only can improve the understandability of the specification and reduce specification errors, but it also plays an essential role in the verification process.

The Signature plugin extends the CoreASM language with syntactic patterns to declare universes, enumerated backgrounds, and function signatures. The corresponding nodes in the parse tree are processed by the Signature plugin when the CoreASM engine is initializing the Abstract Storage (see *Initializing State* in Figure 3.5). During this phase, the engine queries plugins for their contributions to the vocabulary of the state (see definition of InitAbstractStorage in Section 4.5). When the Signature plugin is asked for its vocabulary contribution, it processes the parse tree and provides the engine with a list of universes, backgrounds and functions declared in the specification. Thus, the interpretation of Signature plugin declarations directly modifies the initial state of the simulated machine.

Functions

To declare functions, the Signature plugin extends the CoreASM language with the following syntactic patterns:

Signature Plugin

$$\begin{array}{lcl}
\llparenthesis\,\textbf{function}\ x : \texttt{->}\ x_r\,\rrparenthesis & \rightarrow & \mathsf{CreateFunction}(x, \mathit{controlled}, \langle\rangle, x_r) \\
\llparenthesis\,\textbf{function controlled}\ \ x : \texttt{->}\ x_r\,\rrparenthesis & \rightarrow & \mathsf{CreateFunction}(x, \mathit{controlled}, \langle\rangle, x_r) \\
\llparenthesis\,\textbf{function static}\ \ x : \texttt{->}\ x_r\,\rrparenthesis & \rightarrow & \mathsf{CreateFunction}(x, \mathit{static}, \langle\rangle, x_r)
\end{array}$$

$$\llparenthesis\,\textbf{function}\ x : x_{d_1}\texttt{*}\ \ldots\texttt{*}\ x_{d_n}\texttt{->}\ x_r\,\rrparenthesis \rightarrow \mathsf{CreateFunction}(x, \mathit{controlled}, \langle x_{d_1}, \ldots, x_{d_n}\rangle, x_r)$$

$$\llparenthesis\,\textbf{function controlled}\ \ x : x_{d_1}\texttt{*}\ \ldots\texttt{*}\ x_{d_n}\texttt{->}\ x_r\,\rrparenthesis \rightarrow \mathsf{CreateFunction}(x, \mathit{controlled}, \langle x_{d_1}, \ldots, x_{d_n}\rangle, x_r)$$

$$\llparenthesis\,\textbf{function static}\ \ x : x_{d_1}\texttt{*}\ \ldots\texttt{*}\ x_{d_n}\texttt{->}\ x_r\,\rrparenthesis \rightarrow \mathsf{CreateFunction}(x, \mathit{static}, \langle x_{d_1}, \ldots, x_{d_n}\rangle, x_r)$$

[5] This section is based on Section 5.2 of George Ma's M.Sc. thesis [62] and Section 3.1 of our previously published paper on \Model Checking CoreASM Specifications" [37].

The interpretation of function declaration patterns is defined by the CreateFunction rule, which creates a new function with a specified name, class, and signature.

Signature Plugin

```
**CreateFunction**$(name, functionClass, domain, range) \equiv$
  **let** $f = new(\textsc{FunctionElement})$ **in**
    $class_{fe}(f) := functionClass$
    **let** $s = new(\textsc{Signature})$ **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 function expression, for n-ary functions. Before the function is created, the expression giving its initial value is evaluated. In the following patterns x_c is either $static$ or $controlled$.

Signature Plugin

```
$(\!|$ **function** $x_c$ $x$ : -> $x_r$ **initially** ${}^{\alpha}\boxed{e}$ $|\!)$ $\rightarrow$ $evaluate(\alpha)$
$(\!|$ **function** $x_c$ $x$ : $x_{d_1}$* ...* $x_{d_n}$-> $x_r$ **initially** ${}^{\alpha}\boxed{e}$ $|\!)$ $\rightarrow$ $evaluate(\alpha)$

$(\!|$ **function** $x_c$ $x$ : -> $x_r$ **initially** ${}^{\alpha}v$ $|\!)$ $\rightarrow$ CreateFunctionWithInitValue$(x, x_c, \langle\rangle, x_r, v)$
$(\!|$ **function** $x_c$ $x$ : $x_{d_1}$* ...* $x_{d_n}$-> $x_r$ **initially** ${}^{\alpha}v$ $|\!)$ $\rightarrow$
                                        CreateFunctionWithInitValue$(x, x_c, \langle x_{d_1}, \ldots, x_{d_n}\rangle, x_r, v)$

**CreateFunctionWithInitValue**$(name, functionClass, domain, range, initialValue) \equiv$
  **let** $f = new(\textsc{FunctionElement})$ **in**
    $class_{fe}(f) := functionClass$
    **let** $s = new(\textsc{Signature})$ **in**
      $sigDomain(s) := domain$
      $sigRange(s) := range$
      $signature(f) := s$
      **if** $initialValue \neq undef$ **then**
        SetFunctionValue$(f, domain, initialValue)$
      **add** $(name, f)$ **to** $pluginFunctions(signaturePlugin)$
```

The SetFunctionValue rule sets the initial value of a function. If the function is not nullary and the specified value is a MAPLEMENT, each key in the map is viewed as an argument list and the value of the function for those arguments is set to the corresponding map value.

Universes and Enumerations

The Signature plugin also extends the CoreASM language with patterns for declaration of universes:

Signature Plugin

$(\!|\, \textbf{universe}\ x \,|\!) \quad \rightarrow \quad \mathsf{CreateUniverse}(x, \{\})$

$(\!|\, \textbf{universe}\ x = \{x_{e_1}, \ldots, x_{e_n}\} \,|\!) \quad \rightarrow \quad \mathsf{CreateUniverse}(x, \{x_{e_1}, \ldots, x_{e_n}\})$

The second pattern 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 members are specified, for each member a static function with the given name is also created.

Signature Plugin

```
**CreateUniverse**$(name, members) \equiv$
  **let** $u = new(\textsc{UniverseElement})$ **in**
    **add** $(name, u)$ **to** $pluginUniverses(signaturePlugin)$
    **forall** $elementName \in members$ **do**
      **let** $e = new(\textsc{Element})$ **in**
        $member_{ue}(u, e) := true$
        **let** $f = new(\textsc{FunctionElement})$ **in**
          **add** $(elementName, f)$ **to** $pluginFunctions(signaturePlugin)$
          $class_{fe}(f) := static$
          $\mathsf{SetValue}_{\mathsf{fe}}(f, \langle\rangle, e)$
```

To declare enumerated backgrounds, the Signature plugin provides the following pattern:

Signature Plugin

$(\!|\, \textbf{enum}\ x = \{x_{e_1}, \ldots, x_{e_n}\} \,|\!) \quad \rightarrow \quad \mathsf{CreateEnumeration}(x, \{x_{e_1}, \ldots, x_{e_n}\})$

The CreateEnumeration rule is similar in spirit to CreateUniverse, as enumerable backgrounds are analogous to static universes. The rule is defined as follows:

Signature Plugin

```
**CreateEnumeration**$(name, members) \equiv$
  let $b$ = $new(\textsc{EnumerationBackground})$ in
    add $(name, b)$ to $pluginBackgrounds(signaturePlugin)$
    forall $elementName \in members$ do
      let $e$ = $new(\textsc{Element})$ in
        $bkg(e) := name$
        add $e$ to $enumMembers(b)$
        let $f$ = $new(\textsc{FunctionElement})$ in
          add $(elementName, f)$ to $pluginFunctions(signaturePlugin)$
          $class_{fe}(f) := static$
          SetValue$_{\mathsf{fe}}(f, \langle\rangle, e)$
```

We model background elements that are defined using the Signature plugin with values of the domain EnumerationBackground. The following function, defined on Enumeration Background elements, holds the set of elements each such background represents:

$$enumMembers : \textsc{EnumerationBackground} \mapsto \textsc{Set}(\textsc{Element})$$

For all $eb \in \textsc{EnumerationBackground}$, we have

- $enumerable(eb)$
  All enumeration background elements are enumerable.
- $enumerate_{EnumerationBackground}(eb) \equiv enumMembers(eb)$

Type Checking on Updates

In order to offer runtime type checking on updates, the Signature plugin extends the control flow of the CoreASM engine by registering for the extension points proceeding the aggregation of updates (see Figure 3.8). We have,

$$\forall em \in \textsc{EngineMode},\ isPluginRegisteredForTransition(signaturePlugin, Aggregation, em)$$
$$pluginExtensionRule(signaturePlugin) = \text{@CheckUpdateSetForTypes}$$

As a result of this registration, when the control flow of the engine moves from the *Aggregation* control state to either *Step Succeeded* or *Step Failed*, the engine calls the CheckUpdateSetForTypes rule of the Signature plugin. This rule goes through the update set and for every update checks the arguments and the value of the update against the signature of the function it is updating and reports the inconsistencies. The following rules formally define this process.

Signature Plugin

```
CheckUpdateSetForTypes ≡
  if $engineProperties$("TypeChecking") = "strict" then
    forall $\langle loc, val, act\rangle \in updateSet$ do
      let $f = stateFunction(state, name_{lc}(loc)), sig_f = signature(f)$ in
        if $sig_f \neq undef$ then
          CheckArguments($args_{lc}(loc), sigDomain(sig_f)$)
          CheckValue($val, sigRange(sig_f)$)

CheckArguments($args, domain$) ≡
  if $|args| \neq |domain|$ then
    Error(`Number of arguments passed do not match the domain of the function.')
  else
    forall $i \in [1..|domain|]$ do
      let $universe = stateUniverse(state, domain(i))$ in
        if $\neg member_{ue}(universe, args(i))$ then
          Error(`Argument does not match the domain of the function.')

CheckValue($v, range$) ≡
  let $universe = stateUniverse(state, range)$ in
    if $\neg member_{ue}(universe, v)$ then
      Error(`Update value does not match the range of the function.')
```

5.4.2 The Scheduling Policies Plugin

The Scheduling Policies plugin provides two basic policies for scheduling of agents by the Scheduler. In any CoreASM specification, the particular scheduling policy to be used can be configured using the CoreASM engine's properties (see also Appendix A.4):

- $pluginSchedulingPolicy(SchedulingPoliciesPlugin) \equiv$
$$\begin{cases} allFirstPolicy, & \text{if } engineProperties(\text{"SchedulingPolicies.Policy"}) = \text{"allfirst"};\\ oneByOnePolicy, & \text{if } engineProperties(\text{"SchedulingPolicies.Policy"}) = \text{"onebyone"};\\ undef, & \text{otherwise.}\end{cases}$$

- $newScheduleRule(allFirstPolicy) \equiv$ @NewSchedule$_{allfirst}$

- $newScheduleRule(oneByOnePolicy) \equiv$ @NewSchedule$_{onebyone}$

All-First Policy

The *all-first* scheduling policy first tries to schedule all the given agents elements together in one batch. Alternative options will be non-deterministic subsets of the given sets of elements. Applied to the scheduling of agents, this policy first suggests the execution of all the agents together and if that fails, it offers various subsets of agents as alternative options.

Scheduling Policies Plugin

```
**NewSchedule**$_{allfirst}(group, set) \equiv$
  **result** := $cons(set, \langle s \mid s \in \mathcal{P}(set) \setminus \{set\} \rangle)$
```

One-by-One Policy

The *one-by-one* scheduling policy provides a schedule that comprises of a series of non-deterministically selected single elements. The policy, however, tries to maintain a \fair" set of schedules over a group by keeping a history of the already scheduled elements and trying to avoid re-scheduling of those elements as long as other non-scheduled elements are still available. Applied to the scheduling of agents in a CoreASM simulation, this policy results in a sequential execution of agents.

Scheduling Policies Plugin

```
**NewSchedule**$_{onebyone}(group, set) \equiv$
  **if** $group \neq undef$ **then**
    **if** $scheduleHistory_{obo}(group)$ = $undef \vee set \setminus scheduleHistory_{obo}(group)$ = $\emptyset$ **then**
      **choose** $e \in set$ **do**
        **result** := $\langle e \rangle$
        $scheduleHistory_{obo}(group)$ := $\{e\}$
    **else**
      **choose** $e \in set$ **with** $e \notin scheduleHistory_{obo}(group)$ **do**
        **result** := $\langle e \rangle$
        **add** $e$ **to** $scheduleHistory_{obo}(group)$
  **else**
    **choose** $e \in set$ **do**
      **result** := $\langle e \rangle$
```

5.4.3 IO Plugin

In an open-system view towards modeling, the system operates in a given environment. The environment affects system runs through actions or events and the system can as well affect the environment by its output. In abstract state machines, the interaction between the system (the machine) and the environment is captured through *monitored* (also called *in*), *shared*, and *out* functions. Monitored functions are controlled only by the environment; they are channels through which the machine observes the environment. In a given state, the values of all monitored functions are determined (and do not change) [20]. Out functions are updated only by the machine and they are read-only for the environment. Shared functions are both controlled and read by the machine and the environment.

The IO Plugin utilizes this machine-environment interaction mechanism of ASM and provides two simple channels of communication between a CoreASM machine and its environment: a **print** rule that outputs values to the environment, and an *input* function to get values from the environment. In both cases, textual representations of values are used.

Functions

To facilitate input from the environment, the IO plugin introduces the following monitored function:

- `input: STRING -> STRING`
  $\mathit{class}_{fe}(\mathit{inputFunction}) = \mathit{monitored}$
  For any given value as its argument, this *input* function queries an input value from the environment (presenting the argument as a prompt or key to the input value). Since this is a monitored function, once its value is set for a certain argument (i.e., message) in a computation step, it will not change before the step is completed.

Rule Forms

To provide an output channel for CoreASM specifications, the IO plugin extends the state of the simulated machine by introducing an `output` function (`output: -> String`) which in any given step holds the output of the previous step. Output values are assigned to `output` by **print** rules. Every **print** rule generates a special update instruction with *printAction* to append the a String element to the value of the `output` function. At the end of each computation step, these special updates will be aggregated into one single update to `output` function.

IO Plugin

$$\llparenthesis\, \mathbf{print}\ {}^{\alpha}\boxed{e}\ \rrparenthesis \rightarrow \quad \mathit{pos} := \alpha$$

$$\llparenthesis\, \mathbf{print}\ {}^{\alpha}v\, \rrparenthesis \rightarrow \quad \begin{array}{l} \mathbf{let}\ l = (\text{“output”}, \langle\rangle)\ \mathbf{in} \\ \quad [\![\mathit{pos}]\!] := (\mathit{undef}, \{\!\!\{\langle l, \mathit{stringElement}(v), \mathit{printAction}\rangle\}\!\!\}, \mathit{undef}) \end{array}$$

Aggregation of Output Messages

In the aggregation phase of every step, print update instructions need to be aggregated into a single regular update to the `output` function. Since the print values are String elements (see Section 5.2.3), and there is no execution order on the print rules that generated these updates, the aggregation of these values can be achieved by concatenation of the values into a single String element in a non-deterministic order. The IO plugin provides the semantics of such aggregation as follows.

IO Plugin

```
$\textsf{Aggregate}_{IO}(uMset) \equiv$
  seq
    result := $emptyString$
  next
    if $regularUpdatesExist$ then
      HandleInconsistentAggregation($l, uMset, ioPlugin$)
    else
      foreach $u \in printActionUpdates$ do
        result := $concatString$(result, $uiValue(u)$)
        $aggStatus(u, ioPlugin)$ := $successful$
where
  $regularUpdatesExist \equiv \exists u \in uMset, uiAction(u) = updateAction \wedge uiLoc(u) = (\text{“output”}, \langle\rangle)$
  $printActionUpdates \equiv \{u \mid u \in uMset \wedge uiAction(u) = printAction \wedge uiLoc(u) = (\text{“output”}, \langle\rangle)\}$
```

Composition of Output Messages

In order to maintain the order of output values in a sequential composition of print updates, the composition algorithm provided by the IO plugin aggregates the output values of the first and second step and concatenates them together into a single print update instruction on the `output` function. The the output values of the first step are only considered if the second step does not have a regular update on the output location.

IO Plugin

```
$\textsf{Compose}_{IO}(uMset_1, uMset_2) \equiv$
  local $outputStr$ [$outputStr$ := $emptyString$] in
    seq
      if $\neg regularUpdatesExist(uMset_2)$ then
        foreach $u \in printUpdates(uMset_1)$ do
          $outputStr$ := $concatString$(result, $uiValue(u)$)
    seq
      foreach $u \in printUpdates(uMset_2)$ do
        $outputStr$ := $concatString$(result, $uiValue(u)$)
    next
      result := $\langle(\text{“output”}, \langle\rangle), outputStr, printAction\rangle$
where
  $printUpdates(mset) \equiv \{u \mid u \in mset \wedge uiLoc(u) = (\text{“output”}, \langle\rangle) \wedge uiAction(u) = printAction\}$
  $regularUpdatesExist(mset) \equiv \exists u \in mset, uiAction(u) = updateAction \wedge uiLoc(u) = (\text{“output”}, \langle\rangle)$
```

5.4.4 Step Plugin

The Step Plugin offers a rule constructs allowing the sequential execution of ASM rules that span over more than one computation step. The idea is to introduce a rule construct of the form

$$R_1 \ \mathbf{step} \ R_2$$

evaluation of which takes two computation steps: in the first step, the result of evaluation is equivalent to evaluating R_1 and in the second step the result is that of evaluating R_2. A computation step of an ASM machine M in state $\mathfrak{A}_M$ and program P_M is achieved by evaluating P_M into a set of updates that will be applied to $\mathfrak{A}_M$. The introduction of the **step** construct is faithful to this definition and is a conservative extension to ASMs, meaning that it does not alter the evaluation of P_M. The evaluation of R_1 **step** R_2 results in a set of updates, alternatively the updates of R_1 and R_2.[6]

In order to define a generic and compositive semantics for this constructs, we first define the following two notions.

1. We define a compound *global control state* for machine M in every state $\mathfrak{A}_M$ as a set of local control states. The current value of the global control state of M in any state $\mathfrak{A}_M$ is kept as the value of a nullary function *ctl_state* in $\mathfrak{A}_M$. The idea is that there can be parallel control flows in a single machine M that advance in every computation step. This view of defining the contorl sate of M as a set of local control state lifts the limitation we have in Control State ASMs that disallows parallel control branches.

2. We define a *unique control state identifier* at runtime (evaluation time) for every rule R_i in any **step** construct that takes into account the macro-rule call path to R_i from P_M (in case R_i is not in the body of P_M and is defined in a macro rule defintion). We assume that for every rule R_i, the unique control state identifier of R_i is given by *uniqueCtlState*(R_i).

[6] It is important to note that in M **step** N, M and N represent two ASM rules and not individual machines.

```
$R_1$ **step** $R_2 \equiv$
    **if** $\mathit{uniqueCtlState}(@R_1) \in \mathit{ctl\_state}$ **then**
      $R_1$
       **seq**
      **if** $\nexists\ cs \in \mathit{ctl\_state}\ \ \mathit{subControlState}(cs, @R_1)$ **then**
         **remove** $\mathit{uniqueCtlState}(@R_1)$ **from** $\mathit{ctl\_state}$
         **add** $\mathit{uniqueCtlState}(@R_2)$ **to** $\mathit{ctl\_state}$
    **else if** $\mathit{uniqueCtlState}(@R_2) \in \mathit{ctl\_state}$ **then**
      $R_2$
       **seq**
      **if** $\nexists\ cs \in \mathit{ctl\_state}\ \ \mathit{subControlState}(cs, @R_2)$ **then**
         **remove** $\mathit{uniqueCtlState}(@R_2)$ **from** $\mathit{ctl\_state}$
    **else**
      **add** $\mathit{uniqueCtlState}(@R_1)$ **to** $\mathit{ctl\_state}$
```

The following patterns formally define the semantics of **step** :

Step Plugin

```
$(\!|\,{}^{\alpha}\boxed{r}_1$ **step** ${}^{\beta}\boxed{r}_2\,|\!)$   $\rightarrow$   **if** $\mathit{uniqueCtlState}(\alpha) \in \mathit{ctl\_state}$ **then**
                                       $\mathit{pos} := \alpha$
                                    **else**
                                       **if** $\mathit{uniqueCtlState}(\beta) \in \mathit{ctl\_state}$ **then**
                                          $\mathit{pos} := \beta$
                                       **else**
                                          **add** $\mathit{uniqueCtlState}(\alpha)$ **to** $\mathit{ctl\_state}$
$(\!|\,{}^{\alpha}u_1$ **step** ${}^{\beta}\boxed{r}_2\,|\!)$   $\rightarrow$      **if** $\nexists cs \in \mathit{ctl\_state}\ \ \mathit{subCtlState}(cs, \mathit{uniqueCtlState}(\alpha))$ **then**
                                          **remove** $\mathit{uniqueCtlState}(\alpha)$ **from** $\mathit{ctl\_state}$
                                          **add** $\mathit{uniqueCtlState}(\beta)$ **to** $\mathit{ctl\_state}$
                                       $[\![\mathit{pos}]\!] := (\mathit{undef}, u_1, \mathit{undef})$


$(\!|\,{}^{\alpha}\boxed{r}_1$ **step** ${}^{\beta}u_2\,|\!)$   $\rightarrow$     **if** $\nexists cs \in \mathit{ctl\_state}\ \ \mathit{subCtlState}(cs, \mathit{uniqueCtlState}(\beta))$ **then**
                                         **remove** $\mathit{uniqueCtlState}(\beta)$ **from** $\mathit{ctl\_state}$
                                     $[\![\mathit{pos}]\!] := (\mathit{undef}, u_2, \mathit{undef})$
```

For now, we can define the global control state of the machine as a monitored function that returns the value of *ctl_state*.

Questions and concerns:

1. How do we want multiple calls to a single macro rule R be handled? I.e., should the execution continue regardless of where R is being called?

2. In every computation step the main program of the machine is evaluated as a whole, so if an **step** construct is guarded by a condition and the condition does not hold in subsequent steps, that local **step** will not continue while the machine continues.

5.4.5 The Observer Plugin

It is sometimes desirable to have a machine-readable log of the execution of a CoreASM specification for offline analysis and visualization. One argument for such a feature is that it allows for a clear separation of the execution and the analysis. For example, execution of certain specifications may be time-consuming, but once the execution is done, visualization of the run of the system can be done more quickly and repeatedly, if all the updates of interest are recorded.

The Observer plugin monitors the execution of specifications in CoreASM and produces an XML log of the updates that are produced after every computation step. The plugin can be configured so that only the updates on certain locations of interest are recorded. In order to monitor the updates, the plugin registers itself for the extension point where the control flow of the engine switches to the *Step Succeeded* control state (see Figure 3.6). We have,

$$\forall s \in \textsc{EngineMode},\ \mathit{isPluginRegisteredForTransition}(\mathit{observerPlugin}, s, \mathit{stepSucceeded})$$

where $\mathit{observerPlugin} \in \textsc{Plugin}$ is the Observer plugin.

At this point in the engine lifecycle (when the control state changes to *Step Succeeded*), the computation step is successfully completed and the updates are applied to the state. The Observer plugin then simply goes through the last set of updates and records an XML log of those updates that modify the locations of interest.

$$\mathit{pluginExtensionRule}(\mathit{observerPlugin}) = @\mathsf{FireOnModeTransition}_{\mathit{Observer}}$$

Observer Plugin

```
FireOnModeTransition$_{Observer}$($sourceMode, targetMode$) $\equiv$
  if $targetMode = stepSucceeded$ then
    local $xmlElement$ [$xmlElement := newStepXMLElement$] in
      seq
        foreach $u$ in $updateSet$ with $uiLoc(u) \in observerLocationsOfInterest$ then
          AddXMLChildElement($xmlElement, newUpdateXMLElement(u)$)
      next
        AppendToLog($xmlElement$)
```

5.4.6 Math Plugin

In writing executable specification, one may need to have access to various mathematical constants (such as π) or functions (such as the trigonometric functions) as part of the Number background. The Math plugin addresses this requirement by extending the vocabulary of CoreASM

- `abs(v)` returns the absolute value of v.
- `asin(v)` returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.
- `floor(v)` returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer.
- `log(v)` returns the natural logarithm (base e) of v.
- `max(v1, v2)` returns the greater of two values.
- `min(v1, v2)` returns the smaller of two values.
- `pow(x, y)` returns the value of the first argument raised to the power of the second argument.
- `powerset(set)` computes the powerset of the given set.
- `sum({v1,...,vn}, @f)` returns the sum of a collection of numbers, after applying function `f` to the values in the collection. If there is one non-number in the collection, it returns *undef*.

An Example

```
CoreASM MathPluginExample

use Standard
use Math

init Init

rule Init = {
    program( self ) := @Main
    a(1) := 5
    a(2) := 10
    a(100) := 500
}

rule Main =
    let e = MathE in {
        print "’e’ = " + e
        print "log(e) = " + log(e)
        print "sin(30) = " + round( sin( toRadians(30) ) * 10 ) / 10
        print "asin(0.5) = " + round( toDegrees( asin(0.5) ) )
        print "min(51, 43) = " + min(51, 43)
        print "sum( 1, 2, 100 ) = " + sum({1, 2, 100})
        print "sum( 1, 2, 100, @a ) = " + sum({1, 2, 100}, @a)
        print "powerset(1, 2, 3) = " + powerset({1, 2, 3})
        print "2, 3 memberof powerset(1, 2, 3 = "
                    + ({2, 3} memberof powerset({1,2,3}))
        choose x in powerset({1, 2, 3, 4}) do
            if x memberof powerset({1, 2, 3}) then
                print x + " is a member of powerset(1, 2, 3)"
            else
                print x + " is not a member of powerset(1, 2, 3)"
        ifnone
            print powerset({1, 2, 3})
    }
```

As an example, the output of the execution of Program ?? is the following:

```
sum( {1, 2, 100} ) = 103
min(51, 43) = 43
asin(0.5) = 30
powerset({1, 2, 3}) = {{}, {3}, {2}, {3, 2}, {1}, {3, 1}, {2, 1}, {3, 2, 1}}
{2, 3} memberof powerset({1, 2, 3} = true
log(e) = 1
{3, 2, 4} is not a member of powerset({1, 2, 3})
sum( {1, 2, 100}, @a ) = 515
'e' = 2.718281828459045
sin(30) = 0.5
```

5.4.7 The Time Plugin

To introduce the notion of time in CoreASM, the Time plugin extends the vocabulary of the state with a nullary monitored function

```
now: -> NUMBER
```

that provides the current time of the system as a numeric value. Although, such a monitored function seems to be all that is basically needed to have the notion of time in CoreASM, future versions of this plugin could introduce various functions to extract date and time components from time values (e.g., day of the week, hours, or minutes) or to produce specific or relative time values, such as *12/May/2009* or *now - two hours*.

5.4.8 Property Plugin

The Property Plugin is a small plugin that allows correctness properties, 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 plugin to check simple global assertions). Correctness properties are only applicable during model checking, and are translated by our CoreASM to Promela translator.

The Property plugin provides the following pattern to declare new LTL properties:

[check] property *LTL-property*

Including the keyword **check** with a property declaration indicates that the property should be checked during model checking.

The following operators are defined with the given precedence levels (see 4.2.4):

operator	precedence level	description
G	500	Always
F	500	Eventually
X	500	Next
U	400	Until
V	400	Release

The Property plugin was developed to improves 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 plugin allows multiple properties to be specified for a single specification.

5.5 The JASMine Plugin

In this chapter we have introduced various CoreASM plugins implementing most common mathematical objects and structures, such as *numbers*, *sets*, *lists*, and *maps*.[7] While these backgrounds are usually sufficient for modeling most algorithms and systems, complex specifications may need more advanced features, not necessarily data-oriented. For example, an executable specification for a new peer-to-peer protocol may need access to *network sockets* and *files*; a specification that is used as an executable stub for a software module that still has to be implemented or for a missing piece of hardware may need to put up an on-screen *window* showing its current state; a complex numerical algorithm which is already specified by some standard may be moved out of a specification and a *concrete implementation* written in a standard programming language may be used in its place.

There is thus a clear need to allow *interaction* between CoreASM specifications and concrete code, including operating systems functions, external libraries, and custom code. Among the various tools for running ASM models [28], AsmL (ASM Language) [66], XASM (eXtensible ASM) [2], and AsmGofer [69] provide some support

[7] This section is based on a joint work with Dr. Vincenzo Gervasi and is currently under publication in [43].

for interaction with external programming languages. AsmL, built on the Microsoft .NET framework [65], incorporates numerous object-oriented features and constructs of Microsoft .NET and supports interaction with external .NET classes. The XASM language allows external C-functions to be used in XASM specifications. However, the arguments and return values of C-functions can only be of a specific C-type that represents elements of the super-universe in XASM. Newer versions of XASM support interaction with Java classes but the support is only limited to invoking Java object constructors. AsmGofer [69], an ASM interpreter embedded in the functional programming language \Gofer", supports the use of functional programming in the definition of types and functions.

In this section, we present JASMine, a CoreASM plugin that offers a solution for the interaction of CoreASM specifications and concrete code by integrating Java with CoreASM.

5.5.1 Requirements and Limitations

The Java Class Library provides an extremely rich (and continuously growing) set of APIs and efficient implementations for almost any computing task. Moreover, Java offers platform-independence, support on a wide variety of architectures, and many modern language features that make it an attractive target for the integration of ASM specifications with concrete code.

However, there is a risk that by intertwining the \ASM world" of elements, functions and predicates and the \object world" of an object-oriented language, the very nature of the ASM paradigm may be changed in fundamental ways. This is, for example, what happened in AsmL [66], where rules and methods, elements and objects, sets and the Set object of the .NET framework become confused.

In contrast to AsmL, we do not want interaction with Java to *pollute* the CoreASM word. In particular,

- we want to maintain typelessness of the language: it must be possible to treat Java objects as regular ASM values, and to pass untyped ASM elements as arguments to Java methods (with type checking performed at run time only);

- we want to maintain the parallel model of execution of ASMs: the notion of *step* must be preserved, as well as the assumption that the ASM state and environment is observed in a stable snapshot, and updates are applied in parallel and only when no conflicts arise;

- we want to avoid the introduction of extraneous fundamental concepts: the notions of *state*, *update* and *step* should suffice to describe the computation.

The fundamental choice of preserving the ASM computation model sets strong constraints on how JASMine works, which will be described later in more detail.

Four basic capabilities are needed for a minimal reasonable level of interaction, namely: 1) instantiating new objects, invoking their constructors, and storing a reference to the new object in the ASM state; 2) accessing (reading and writing) public fields of objects, including static fields of classes; 3) invoking public methods of objects and static methods of classes, passing the needed arguments, and storing the result in the ASM state; 4) converting between certain ASM types and the corresponding Java types and back, as needed to support expression evaluation and updates. The mechanisms we propose to provide these capabilities constitute a *conservative extension* of CoreASM, in the sense that the semantics of the non-JASMine parts of a specification are not altered by the extension[8].

Notice that the integration that JASMine provides between ASMs and Java is far less complete than the one existing between, for example, AsmL and .NET: in particular, it is not currently possible to define new Java classes or interfaces through ASM specifications, nor is it possible to use Java inheritance in CoreASM specifications. Interfaces and abstract classes cannot be accessed at all.

We do not see these limitations as particularly relevant in practice. In fact, the design goal of JASMine is to allow *interaction* between ASMs and Java, rather than full *integration*, and we believe the JASMine plugin serves well in this capacity.

5.5.2 Language Extensions

The following subsections describe in turn the constructs implementing the four capabilities mentioned above.

Creation of Java Objects

Java objects in JASMine are seen as part of the *environment*, not of the *state*. This is a fundamental design choice, which differs from what others have done (e.g., AsmL), and helps in cleanly separating the structures-based state of ASM, which only changes between steps and through non-conflicting updates, from the independently evolving state of Java, which can change at any time and also due to external events (e.g., a timer or GUI actions).

JASMine introduces a new background (hence, a new kind of element in the ASM state) called `JObject` which holds a *reference* to the real Java object. Only this immutable reference enters the ASM state as a value, and only through a special update command, hence the basic ASM computation cycle is preserved. As a consequence, creation of a new object is not considered an expression (as is the `new` operator in Java) but rather a rule, since it results in an update. We have

$$\begin{gathered} jObjectBack \in \textsc{BackgroundElement} \\ name(jObjectBack) = \text{“JOBJECT”} \\ newValue(jObjectBack) = newJObject() \end{gathered}$$

[8] In other terms, a specification which does not interact with Java, and thus does not use the JASMine constructs, has the same semantics whether it includes the JASMine plugin or not.

where *newJObject*() returns a new `JObject` element pointing to a new Java object.

In formal terms, using the notation described above, creation of a new Java object is accomplished as follows:

```
                                                                  CreationRules
$(\!|\,\textbf{import native } {}^{\alpha}\square \textbf{ into } {}^{\beta}\boxed{l}\ |\!) \rightarrow \quad pos := \beta$
$(\!|\,\textbf{import native } {}^{\alpha}x \textbf{ into } {}^{\beta}l\,|\!) \rightarrow$  if $isJavaClassName(x)$ then
                                   if $hasEmptyConstructor(x)$ then
                                      EvaluateImport$(l, x, \langle\rangle)$
                                   else
                                      Error(`Constructor not found.')
                                else
                                   Error(`Java class not found.')

$(\!|\,\textbf{import native } {}^{\alpha}x({}^{\lambda_1}\boxed{e}_{\,1}, \ldots, {}^{\lambda_n}\boxed{e}_{\,n}) \textbf{ into } {}^{\beta}\boxed{l}\ |\!) \rightarrow \quad pos := \beta$
$(\!|\,\textbf{import native } {}^{\alpha}x({}^{\lambda_1}\boxed{e}_{\,1}, \ldots, {}^{\lambda_n}\boxed{e}_{\,n}) \textbf{ into } {}^{\beta}l\,|\!) \rightarrow$
        if $isJavaClassName(x)$ then
           choose $i \in [1..n]$ with $\neg evaluated(\lambda_i)$
              $pos := \lambda_i$
           ifnone
              if $hasConstructor(x, \langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle)$
                 EvaluateImport$(l, x, \langle\lambda_1, \ldots, \lambda_n\rangle)$
              else
                 Error(`Constructor not found.')
        else
           Error(`Java class not found.')
```

Here, we use the *jValue* function to abstract from the task of potentially converting CoreASM elements to Java objects (see Page 134). The actual evaluation of the **import native** statement is defined by the following macro, which takes as parameters a location where to store the reference to the new Java object (as a JObject value), an identifier representing the name of the class, and a sequence of positions of values, which will be the actual parameters for the constructor call:

```
                                                                  EvaluateImport
EvaluateImport$(l, x, \langle\lambda_i, \ldots, \lambda_n\rangle) \equiv$
   let $u$ = DefUpd(CREATE, $(l, x, \langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle)$) in
      let $jtl$ = (“jasmChannel”, $\langle\rangle$) in
         $[\![pos]\!] := (undef, \{\!|\langle jtl, u, jasmAction\rangle|\!\}, undef)$
```

Notice in the specification above how the execution of the rule does not really instantiate the new object (whose constructor could have side effects, and thus alter the Java state), but instead accumulates a special update instruction (a *deferred update*) akin to the update instructions used for aggregation and partial updates [64]. Actual instantiation will be performed at update application time, as will be shown later on. The designated location (“jasmChannel”, $\langle\rangle$) accumulates all the JASMine-related update instructions that are performed during a step, whereas the DefUpd macro produces an encoding of its parameters, suitable for later execution of the relevant update.

While the subject will be discussed more fully in the following, it is worthwhile to remark here that this strategy ensures that any action that can perturb the environment (e.g., instantiation of a new Java object) will only be taken if the step turns out to be effective, i.e. if no conflicting updates are generated in that step.

Access to Fields of Java Objects

Reading a field in a Java object does not have side effects and thus can be treated as a pure expression as far as the ASM computation cycle is concerned[9]. In particular, the value in the field can be computed immediately at expression evaluation time. In contrast, writing into a field has observable side effects, and thus cannot be performed *during* a step, but only *between* steps; the corresponding value is then stored in the field at update application time through another deferred update. The following rules detail the semantics used for field access in JASMine.

FieldReadExpression

```
$\llparenthesis {}^{\alpha}\boxed{e}\ \texttt{->}{}^{\beta}x \rrparenthesis$    $\rightarrow$    $pos := \alpha$
$\llparenthesis {}^{\alpha}v\texttt{->}{}^{\beta}x \rrparenthesis$    $\rightarrow$    if $isJObject(v)$
                                  if $hasField(jObj(v), x)$
                                    if $ImplicitConversionMode$ then
                                      $[\![pos]\!] := (undef, undef, asmValue($GetField$(jObj(v), x)))$
                                    else
                                      $[\![pos]\!] := (undef, undef, newJObject($GetField$(jObj(v), x)))$
                                  else
                                    Error(`No such field.')
                                else
                                  Error(`Not a Java object.')
```

As can be observed, field access expressions are evaluated by first evaluating the reference to the JObject, and then (after checking that the given value is actually a JObject and that the corresponding class has an accessible field with the given name) the value in the field of the Java object is retrieved, possibly converted to its ASM counterpart based on the configuration of the plugin (see Section 5.5.2), and finally used as the value of the whole expression. Access to static class fields are handled similarly, and we skip here the corresponding rules for brevity.[10] Assignments are treated through deferred updates:

[9] In a multi-threaded context, field values can change at any moment, even without any write action by the ASM specification. To guarantee the stability of the environment, values read from Java fields are cached by JASMine when first read, and the same value is used if the same field read expression on the same Java object is evaluated multiple times in the same step.

[10] Reading a static field of a class that has a static block and is not initialized can potentially have side effects. Currently, we do not handle this special case and treat static fields and object fields the same with regard to read access.

FieldWriteRule

```
$(\!|$ store $^{\alpha}\boxed{e}$ into $^{\beta}\boxed{e}$ ->$^{\gamma}x|\!) \rightarrow$
    choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
      $pos := \lambda$
    ifnone
      if $isJObject(value(\beta))$ then
        if $hasField(jObj(value(\beta)), x)$ then
          let $u$ = DefUpd(STORE, $(value(\beta), x, jValue(value(\alpha)))$) in
            let $jtl$ = (“jasmChannel”, $\langle\rangle$) in
              $[\![pos]\!] := (undef, \{\!|\langle jtl, u, jasmAction\rangle|\!\}, undef)$
        else
          Error(`No such field.')
      else
        Error(`Not a Java object.')
```

Notice how write access to fields is treated as a partial update to the internal structure of the JObject element. Before the engine applies the updates to the state, the JASMine plugin as the corresponding aggregator will have to check that no conflicting assignments to the same field of a given JObject element are performed, and moreover that the JObject as a whole is not updated to a different value in the same step[11]. Once more, write access to static fields of classes is very similar and we do not detail it here.

Invoking Methods of Java Objects

As remarked above, invocation of methods in Java objects can have side effects which can change both the internal state of the object and of other objects as well (i.e., by calling other methods or accessing public fields). For this reason, method invocation is handled through a deferred update, as described below. Two forms of method invocation exists: one for *void* methods, which have no return value, and one for methods returning a value. The simplest version for void methods invocation is specified as follows:

[11] The same situation is found in other cases, e.g. when both $a := \{1, 2\}$ and **add** 3 **to** a appear in the same step.

VoidMethodInvocationRule

```
$(\!|\,\textbf{invoke}\ {}^{\alpha}\boxed{e}\ \texttt{->}^{\beta} x({}^{\lambda_1}\boxed{e}_{\,1}, \ldots, {}^{\lambda_n}\boxed{e}_{\,n})\,|\!) \rightarrow$
    $\textbf{choose}\ \lambda \in \{\alpha, \lambda_1, \ldots, \lambda_n\}\ \textbf{with}\ \neg evaluated(\lambda)$
      $pos := \lambda$
    $\textbf{ifnone}$
      $\textbf{if}\ isJObject(value(\alpha))$
        $\textbf{if}\ hasMethod(jObj(value(\alpha)), x, \langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle)$
          $\textbf{let}\ u = \text{DefUpd}(\text{INVOKE},$
                    $(undef, value(\alpha), x, \langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle))\ \textbf{in}$
            $\textbf{let}\ jtl = (\text{“jasmChannel”}, \langle\rangle)\ \textbf{in}$
              $[\![pos]\!] := (undef, \{\!|\langle jtl, u, jasmAction\rangle|\!\}, undef)$
        $\textbf{else}$
          Error(`No such method.')
      $\textbf{else}$
        Error(`Not a Java object.')
```

The version for non-*void* methods is only slightly more complex. We provide a special update instruction (in the vein of **add** ... **to** ...) so that the actual method call is only performed if the update set is guaranteed to be consistent (see section 5.5.2 for detailed conditions).

This solution may be inconvenient at times. For example, it is not possible to assign directly the result of a method invocation to a field of the same or of a different object, as two separate **invoke** and **store** instructions are needed, and in two different steps. In other words, the effect of any rule altering the state of the \Java world" is only observable in the *next* step of the machine, which of course discourages programming in a sequential style: instead, any needed sequentiality will have to be made explicit, e.g. by using an FSM representation of the ASM. Also, field updates and method invocations performed in the same step will be performed | in due time | in an unspecified order, since update instructions in CoreASM constitute an unordered multiset. This behavior, too, may surprise the unaware Java programmer at his first approach with ASMs, as will be discussed in Sections 5.5.4 and 5.5.5.

Nevertheless, we believe that the soundness of the semantics that is given by the deferred updates approach is worth the inconvenience, and can actually help even novice specifiers in drawing a clear line between what needs to be specified and the actual behavior (possibly, over-specified) of the implementation.

Formally, invocation of non-*void* methods is specified as follows:

NonVoidMethodInvocationRule

```
$\llparenthesis$ invoke $^{\alpha}\boxed{e}$ ->$^{\beta}x(^{\lambda_1}\boxed{e}_1, \ldots, ^{\lambda_n}\boxed{e}_n)$ result into $^{\gamma}\boxed{l}$ $\rrparenthesis \rightarrow$
        choose $\lambda \in \{\alpha, \gamma, \lambda_1, \ldots, \lambda_n\}$ with $\neg evaluated(\lambda)$
          $pos := \lambda$
        ifnone
          if $isJObject(value(\alpha))$
            if $hasMethod(jObj(value(\alpha)), x, \langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle)$
              if $loc(\gamma) \neq undef$
                let $u$ = DefUpd(INVOKE, $(loc(\gamma), value(\alpha), x,$
                                    $\langle jValue(value(\lambda_1)), \ldots, jValue(value(\lambda_n))\rangle))$ in
                  let $jtl$ = ("jasmChannel", $\langle\rangle$) in
                    $[\![pos]\!] := (undef, \{\!\{\langle jtl, u, jasmAction\rangle\}\!\}, undef)$
              else
                Error(`Cannot update a non-location.')
            else
              Error(`No such method.')
          else
            Error(`Not a Java object.')
```

As for the previous constructs, we do not detail here how static methods on classes are invoked, as the mechanism is totally analogous.

In practice, if an exception is returned, two updates are produced: one storing the value of the exception (as an ASM JObject) in a designated location, and another one storing a different value to the same location. As a consequence, Java exceptions are mapped in ASMs to conflicting updates, which can be caught via the Turbo ASM **try/catch** rule [20].

Type Conversion

JASMine operates in two type conversion modes: *implicit conversion* and *explicit conversion*. In the implicit mode, which is the default mode, JASMine automatically converts types between CoreASM and Java when needed. This reduces the hassle of type conversion and helps in writing more concise CoreASM specifications. Automatic type conversion, however, has its drawbacks in certain applications: it converts values even when such a conversion is not needed; e.g., when returned values of Java methods are to be passed as arguments in future calls to other Java methods. In the explicit mode, the user is responsible for explicitly converting values between Java and CoreASM using the provided CoreASM functions described further below.

JASMine constructs apply type conversion when needed, through the functions *javaValue* and *asmValue* that convert CoreASM values to Java objects and vice versa. These two functions are defined by cases as summarized in Table 5.1. In most of the rules presented in this paper, the *jValue* function abstracts the details of type conversion based on the conversion mode.

The JObject background offers the following two functions, which perform the same conversion on arbitrary values:

Java type	**CoreASM background**
bool, Boolean	Boolean
byte, short, int, long, float, double, Byte, Short, Integer, Long, Float, Double	Number
char, Character	currently not supported
String	String
Set interface	Set
List interface	Sequence
Map interface	Function (dynamic)
arrays	currently not supported
any other object	JObject

Table 5.1: Type Conversions Between CoreASM and Java.

- `toJava: Element -> JObject`
  $value_{fe}(toJavaFunction, \langle v \rangle) = javaValue(v)$
- `fromJava: JObject -> Element`
  $$value_{fe}(fromJavaFunction, \langle v \rangle) = \begin{cases} asmValue(jObj(v)), & \text{if } isJObject(v); \\ \mathsf{undef}_e, & \text{otherwise.} \end{cases}$$

Aggregation of Deferred Updates

As we have seen, any modification to the \Java world" is performed through special update instructions, called deferred updates (but not to be confused with ASM updates), to ensure a stable state and a stable environment in course of a single ASM computation step. Three types of deferred updates are used by JASMine: instantiation (CREATE), field writing (STORE) and method invocation (INVOKE).

Each type of deferred update carries the information necessary for its execution; in particular, CREATE carries information on the Java class to create and on the location of the new ASM element to create; STORE carries information about the JObject whose field is to be modified, about the name of the field to modify, and about the new value to be written in the field; INVOKE carries information about the JObject on which the method has to be invoked, about the name of the method, and about the (possibly empty) list of arguments to pass to the method.

The following compatibility conditions must be met for a set of updates to be considered consistent:

1. No other update is permitted on the ASM location used in a CREATE. Notice that this includes JASMine deferred updates (i.e., it is not possible to import twice to the same location) as well as regular updates (i.e., it is not possible to assign a different value through the assignment operator := or other update rules to a location used in a CREATE).

2. If multiple STOREs are performed on the same field of the same object, they must all assign the same value.

3. Any location used to store the result of an INVOKE cannot appear in any other update.

Notice that this latter condition is sufficient, but not necessary to guarantee consistency. In fact, we disallow even multiple updates that would write the same value (which are normally permitted under standard ASM semantics). The reason for this more restrictive choice is that in general it is impossible to know which value will be returned by a method call without actually calling the method, and we want the method to be called only if a consistent set of updates is generated. Hence, we require a stronger guarantee than what is strictly needed.

If the set of update instructions is consistent, the prescribed operations are performed *in unspecified order*. Notice that the first condition above ensures that newly-created JObjects are not used in the same step, so there is no need to specify a special ordering with CREATE update instructions performed before STORE and INVOKE ones.

A common troublesome case is when multiple method invocations are performed: if the particular sequence is order-sensitive, ordering will have to be specified explicitly by using a finite state automaton. In most cases, though, the specific order will be immaterial (e.g., `Point.setX()` and `Point.setY()`), and in these cases multiple invocations can well be specified in the same step. We regard this as a desirable feature for a specification: in fact, the implementer will know that fields can be written and that methods can be invoked in any order as long as they are specified to happen in a single ASM step, whereas the ordering between different steps is significant, and should be respected in the implementation.

5.5.3 Implementing JASMine

In its capacity as a bridging technology, JASMine has to interact closely with both the CoreASM engine and the Java virtual machine. We will discuss these interactions in the following.

Interacting with the CoreASM Engine

The CoreASM extensibility architecture dictates that plugins extending the basic CoreASM language have to implement one or more interfaces, depending on which elements of the language (both syntax and semantics) and of the computation cycle are contributed. In particular, JASMine provides the following extensions:

- It implements the *parser plugin* interface to extend the parser with new syntax for native import, field read/write, and method invocation. The syntax rules contributed to the language correspond to the syntactical patterns shown in Section 5.5.1.

- It implements the *interpreter plugin* interface and contributes the semantics for the new syntactical patterns. The semantics contributed correspond to the ASM rules shown in Section 5.5.1.
- It implements the *vocabulary extender* interface to extend the CoreASM state with the JObject background and the monitored *jasmChannel* function. In particular, the two casting functions `toJava` and `fromJava` are introduced as part of the JObject background. Moreover, element equality, ordering and conversion to a String value are forwarded to the Java object represented by any given JObject value.
- It implements the *aggregator* interface to provide aggregation rules which encode all the JASMine update instructions computed in one step into one single update to the *jasmChannel* location.
- To actually communicate with the Java virtual machine, the value of *jasmChannel* must be read after every successful step and the actions encoded therein must be parsed and applied to the corresponding Java objects. To perform this, the JASMine plugin extends the lifecycle of the CoreASM engine and reads the value of *jasmChannel* whenever the control state of the engine is switched to *Step Successful*, i.e. whenever a step is completed with a consistent set of updates; it then executes all the CREATE, STORE and INVOKE operations stored in *jasmChannel*.

Interacting with the JVM

Interaction between JASMine and the Java Virtual Machine is limited to a few, well-defined operations, and is mostly mediated by the Java Reflection API [72].

The application of updates encoded in *jasmChannel* entails the following steps.

1. For CREATE updates, the classical `Class.forName()` method is invoked, passing a string representation of the imported class name. Once a `Class` object for the desired class is obtained, if the nullary version of `import native` was used (i.e., with no arguments passed to the constructor of the object), the `Class.newInstance()` method is invoked to obtain the instance. Otherwise, `Class.getConstructor()` is called to retrieve the corresponding constructor, then the constructor's `newInstance()` method is called, with the given arguments, to obtain the instance. A new JObject element encapsulating the new instance is then created and assigned to the ASM location provided in the CREATE record.
2. For STORE updates, the class of the referenced object is obtained by calling `getClass()` on the reference held by the JObject; the `Field` object is then retrieved through `Class.getField()`, and finally `Field.set()` (or one of its primitive type variants) is called to assign the value from the STORE record.

3. For INVOKE updates, the class of the referenced object is obtained as above, then the matching `Method` object is retrieved through `Class.getMethod()` (notice that in this way only public methods can be retrieved), and finally `Method.invoke()` is called, with the appropriate parameters from the INVOKE record. If the method was non-void, the resulting value is then stored in the ASM location provided in the INVOKE record.

It is worthwhile to remark that fields and methods name resolution is entirely delegated to the Reflection API, and thus follows the normal resolution algorithm in Java (see [47, sections 8.2 & 8.4]).

Evaluation of field read access is performed immediately upon encountering the corresponding expression, by first obtaining the `Field` object as for STORE updates, then invoking `Field.get()` (or one of its primitive types variants) to retrieve the field value, which is then returned as the expression's value. These operations constitute the GetField macro used in the semantics (Section 5.5.2).

The various functions used in Section 5.5.2 ($isJavaClassName$, $hasEmptyConstructor$, $hasConstructor$, $hasField$, $hasMethod$) are directly mapped to the corresponding Reflection API methods. All these predicates are implemented by trying to access the given class, constructor, field or method and possibly catching the various exceptions (`ClassNotFoundException`, `NoSuchMethodException`, `NoSuchFieldException`) thrown by the Reflection API methods. The $jObj$ function returns a reference to the Java object encapsulated by a JObject.

Finally the conversion functions $javaValue$ and $asmValue$ are implemented by cases, as summarized in Table 5.1. In particular, when converting from CoreASM elements to Java values ($javaValue$ function), Booleans and numbers are simply converted to the corresponding primitive types in Java; numbers are generally converted to double, then downcast as needed to fit smaller types. CoreASM's strings are wrappers around Java strings, so the conversion is trivial. More complex mathematical structures (e.g., set or sequences) are generally implemented in CoreASM as wrappers to the various Java Collections API objects, so in this case also conversion amounts to unwrapping the underlying object. Any other CoreASM value is upcast to `Object` and passed as-is, thus realizing an opaque container for the ASM value from the point of view of Java code.

Conversion from Java values to CoreASM elements ($asmValue$ function) is similar, except that any unrecognized Java object is wrapped in an opaque JObject element from the point of view of ASM code. This allows access to fields and invocation of methods of objects returned from other Java methods, as in

```
invoke calendar->getCurrentDate() result into today
```

followed, in a subsequent step, by

```
wday := today->weekDay
invoke today->add(7) result into nextWeek
```

5.5.4 A Simple Example

In this section, we present a simple example of an ASM using JASMine constructs. Our example, presented below executes in three steps (distinguished by the `mode` function ranging from 1 to 3) and demonstrates the employment of the sorting capabilities of the standard Java library.

```
CoreASM JASMineExample

use Standard
use Jasmine

function mode:  -> NUMBER initially 1

init InitRule

rule InitRule = {
    case mode of
        1:  import native java.util.TreeSet([8, 10, 4, 32]) into list

        2:  {
            print "The list is " + list
            invoke list->size() result into s
            invoke list->add(15)
        }

        3:  {
            print "Size of list is " + s
            print "After adding 15, the list is " + list
        }
    endcase
    mode:= mode + 1
}
```

In the first step, we instantiate a `SortedSet` Java object based on a CoreASM list element. Here, JASMine automatically converts the CoreASM list (and all its elements) into their equivalent Java objects. In the second step, three tasks are done in parallel: the resulting `SortedSet` Java object is printed out, its size is retrieved and stored in a CoreASM location (by invoking its `size()` method), and a new value (15) is added to the list. In the last step, the size of the list and its new value (after adding 15) is printed out. Here is the output of execution:

```
The list is [4, 8, 10, 32]
Size of list is 4
After adding 15, the list is [4, 8, 10, 15, 32]
```

Notice that the values of the list are automatically sorted in the `SortedSet` Java object and the order is maintained even after the addition of `15`. It is also interesting to note that since the addition of `15` is done in parallel with retrieving the size of the list, different runs of the specification may result in either of the values 4 or 5 for the size of the list in the output, depending on in which order these two method calls (`size()` and `add(15)`) are performed by JASMine.

5.5.5 Final Remarks

As we mentioned earlier, in defining the semantics of JASMine we have chosen to be faithful to the theoretical ASM model. This choice has important pragmatic implications that we discuss here.

In particular, JASMine presents a *stable view of the Java environment* to ASMs. This is required by ASM semantics, but may be inconvenient in practice, as any action performed on a Java object (e.g., storing a value in a field or invoking a method) will produce observable effects only in the *next* step of the machine: thus, many programming patterns typical of sequential programming cannot be applied. This is also true in the case of Turbo ASM rules: hence, the n-th step in a **seq** or **iterate** rule will *not* observe the effects on the environment of the previous $n - 1$ steps, as the corresponding updates are being deferred as described in Section 5.5.2. This is due to the impossibility of rolling back the Java environment to a previous state, which prevents speculative execution of the inner steps of a Turbo ASM step. For example, a **while** cycle like

```
import native java.io.File into file
...
while (lastModified <= lastActed)
    invoke file->lastModified() result into lastModified
...
```

which could be used to wait for a modification to a file, will not work as expected: in fact, invocations to `lastModified()` will be deferred until the end of the step, most probably defeating the programmer's intention.

In terms of style, one could argue that such behavior should be either encapsulated inside a single Java method `waitModification()` (to be invoked through JASMine), or lifted up to the top level of the ASM specification.

Chapter 6

Implementing CoreASM

As we addressed in Section 1.2, one of the requirements of the CoreASM modeling environment is that it should be implemented as an open framework, under an open source license, and using a platform-independent programming language, so that it can be later improved or modified as needed by its community of users. Realizing this requirement, we decided to implement CoreASM using the Java programming language, one of the most popular platform-independent[1] programming languages available.

In order to make CoreASM and its source code freely available for both the academic environment and the industry, we had to carefully choose an open source license that provides users and developers the freedom they need to use and modify CoreASM, without the restrictions that come with many open source licenses. After considering various open source licenses such as GNU Lesser General Public License (LGPL) [40], Apache Software License [41], and BSD licenses [68] and looking at similar open source projects, we have decided to make CoreASM source code available under the Academic Free License (AFL) version 3.0[2]. AFL 3.0 is an open source license with no reciprocal obligation to disclose source code; i.e., derivative works can be licensed under other licenses, and the source code of those derivative works need not be disclosed. Such a license provides a good compromise between the availability of the original source code in a free form and the existence of potentially proprietary editions and extensions in the industry.

Currently, the CoreASM project is publicly available on Sourceforge.net,[3] one of the most popular repositories of open source software offering online resources for open source software development and content creation. Since its first beta release in September 2006, CoreASM has gone through a number of revisions and its latest version (under testing at the time of writing this document) offers substantial

[1] According to Java's download page on `http://java.sun.com`, its standard edition is available on a wide variety of hardware and software platforms: Linux, Linux Intel Itanium, Linux x64, Solaris SPARC, Solaris x64, Solaris x86, Windows, Windows Intel Itanium, and Windows x64.

[2] `http://www.opensource.org/licenses/afl-3.0.php`

[3] `http://www.sourceforge.net`

improvements over its previous version in terms of both features and performance.

The rest of this chapter continues with an overview of the architecture of CoreASM in Java. Section 6.2 looks into the implementation of the CoreASM engine focusing on the implementation of the two more challenging components, the Abstract Storage and the Parser, and the implementation of CoreASM plugins. Section 6.3 concludes this chapter by introducing the tools and user interfaces that are built around the CoreASM engine.

6.1 The Architecture

The CoreASM engine has a micro-kernel architecture. Recalling the architecture of CoreASM as presented in Chapter 3, the kernel of the engine provides only the essential aspects of the engine required for the plugins and applications to be built upon. Furthermore, the kernel is decomposed into four components: a parser, an interpreter, an abstract storage, and a scheduler. The interface of the engine to its environment (and in parts, to its four components) is provided by a special component called the Control API (see Figure 3.2).

Closely following the design of the engine, the Java implementation of CoreASM implements the kernel of the engine in terms of four components and a Control API. The interface of the components are defined by four Java interface files: `Parser`, `Interpreter`, `AbstractStorage`, and `Scheduler`. For every component, a default implementation is provided in form of a Java class file. However, every component is carefully encapsulated in its interface and, as a result, a different implementation can be used as long as it complies with the the interface of the component and its specification. Since Control API acts as a double interface, providing services both to the environment of the engine and to its internal components | the former being a subset of the latter, two Java interface files together define the interface of the engine: (i) a `CoreASMEngine` interface defines the interface of the engine to its outside environment offering services such as loading, parsing, or execution of specifications; (ii) a `ControlAPI` which extends the `CoreASMEngine` interface providing access to every component, a mapping of plugin names to actual plugin instances, and error reporting services. An implementation of the CoreASM engine is provided by the Java class file `Engine` which implements the `ControlAPI` interface.

The `CoreASMEngine` interface provides a comprehensive interface to the engine. Through this interface, applications can (i) load CoreASM specifications into the engine, execute them step by step, and access the simulated state and the latest update set throughout the execution, (ii) use the engine as a parser to just parse specifications into parse-trees (which can then be externally processed for various purposes such as model checking [37, 62]), or access the list of plugins required by a given specification, (iii) modify various engine properties and also observe the behavior of the engine by implementing the `EngineObserver` interface.

There are currently two user interfaces available for CoreASM (see Figure 6.1): a

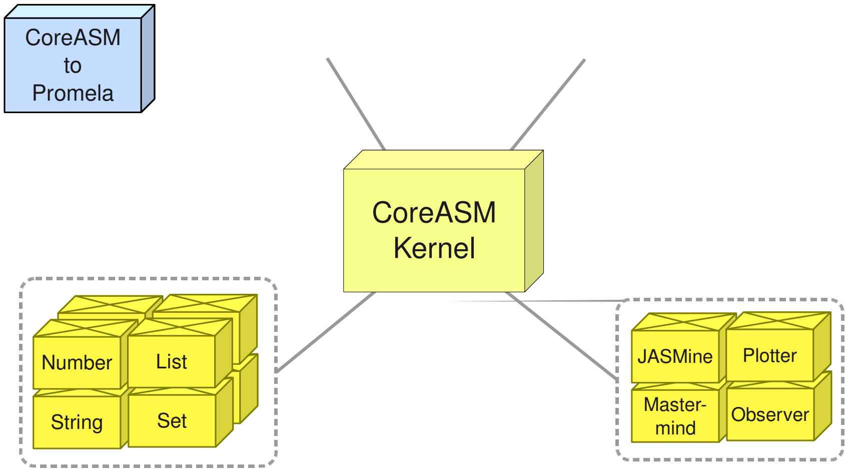


Figure 6.1: CoreASM Kernel, Plugins, and Applications

comprehensive command-line user interface, called Carma, and a graphical interactive development environment in the Eclipse platform, known as the CoreASM Eclipse Plugin. There is also a sophisticated tool under development for creating and modifying Control State ASMs and translating them into CoreASM specifications, called CSDe. Section 6.3 presents these tools in more detail.

The CoreASM kernel also defines the skeleton of a CoreASM plugin in form of a Java abstract class `Plugin`. Various types of extensions that plugins can provide to the engine, such as parser extension or vocabulary extension (see Section 4.5 for a complete list), are defined in terms of Java interface files. Every CoreASM plugin must extend the `Plugin` abstract class and most likely implement one or more of the extension interfaces to offer its contribution to the engine.

6.2 The CoreASM Engine

In this section we briefly look into the implementation of the kernel (focusing on the more challenging components, the Abstract Storage and the Parser) and the plugin framework.

6.2.1 The Kernel

CoreASM engine is represented by the `CoreASMEngine` interface and is implemented by the `Engine` class file which serves two purposes: (i) it provides an implementation for the interface of the engine to its outside environment, and (ii) it acts as a container for the main components of the engine and maintains the control state of

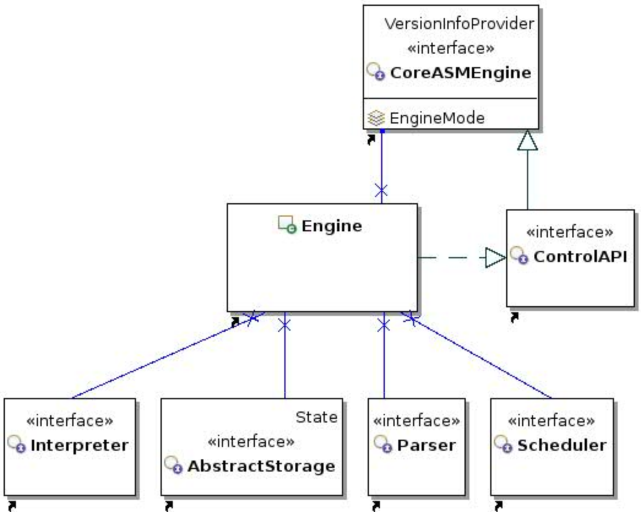


Figure 6.2: Components of the CoreASM Engine

the CoreASM engine. In order for the engine to be always responsive to its environment, the `Engine` object runs in two parallel processing threads: one, being the environment or the *caller*'s thread, responds to requests from the environment (such as sending commands, setting engine properties, or retrieving updates) and the other one maintains the internal control flow of the engine.

The Abstract Storage

The Abstract Storage is implemented by more than three dozen classes in the package `org.coreasm.engine.absstorage`. A hierarchy of classes implement various types of elements defined in the kernel (see Figure 6.3). At the root of this hierarchy, we have the `Element` class which is the superclass of all the values in CoreASM states, implementing the Element domain. Following the specification of Section 4.1, every instance of `Element` has a background and a notion of equality. Three immediate subclasses `BooleanElement`, `RuleElement`, and `FunctionElement` respectively implement the domains of BooleanElement, Rule, and FunctionElement defined in Section 4.1. The domain of BackgroundElement and UniverseElement are implemented by similarly named subclasses of a more generic class `AbstractUniverse` which captures similar aspects of these two domains. Since only a finite set of elements can be represented by Universe elements, `UniverseElement` also implements the `Enumerable` interface.

The main class of this package is `HashStorage`, which offers an implementation for the Java interface `AbstractStorage` based on hash tables. The CoreASM state is implemented by the Java class `HashState` through three separate mappings of names (Java `String` values) to Function elements (instances of `FunctionElement`), Rule elements (instances of `RuleElement`), and Background and Universe elements

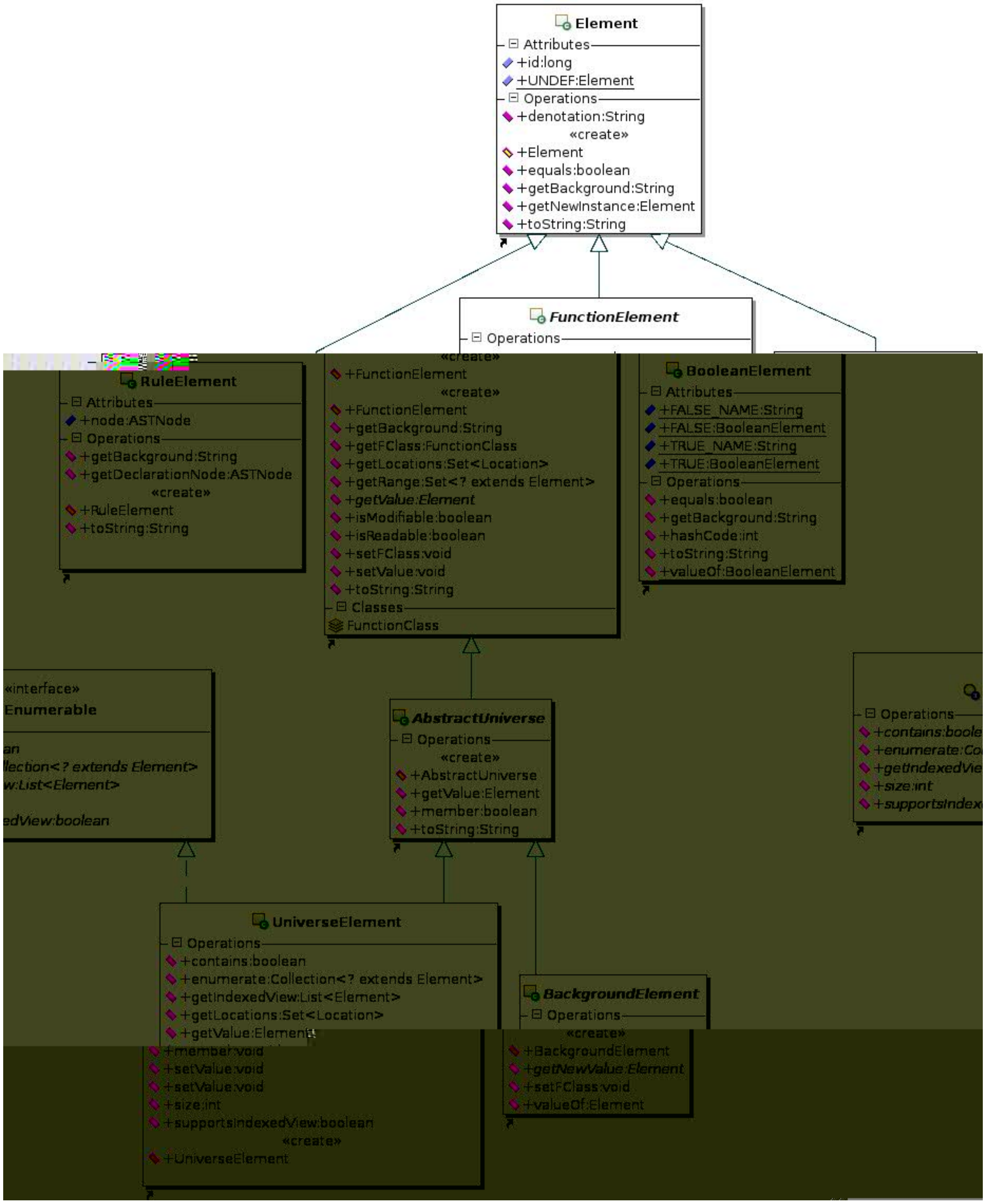


Figure 6.3: Core Elements Defined in the Abstract Storage

(instance of `AbstractUniverse`), thereby implementing contents of CoreASM state as defined in Section 4.1:

$$
\begin{aligned}
&\mathit{stateFunction} : \textsc{State} \times \textsc{Name} \mapsto \textsc{FunctionElement}\\
&\mathit{stateRule} : \textsc{State} \times \textsc{Name} \mapsto \textsc{Rule}\\
&\mathit{stateUniverse} : \textsc{State} \times \textsc{Name} \mapsto \textsc{UniverseElement}
\end{aligned}
$$

The Parser

Implementing the parser component of the CoreASM engine was quite a challenge. At first, we were looking for fast and efficient parser generators that can be called upon loading a specification to generate a parser based on the grammar provided by the specific plugins that are used in that specification. Originally, we used the OOPS (Object Oriented Parser System) parser generator[4] developed and maintained by Axel-Tobias Schreiner and his students Bernd Kühl and William Leiserson. The original OOPS parser generator was quite restrictive for CoreASM as it would generate only LL(1) parsers. Later, Will Leiserson extended and improved OOPS into an LL(k) parser generator [60]. However, the new parser generator was not fast enough on typical CoreASM specifications to be used every time a specification is being loaded.

We looked into a number of available open source parser generators in search of an efficient LL(k) parser generator written in Java and we eventually found jparsec,[5] a recursive-descent parser combinator framework written for Java. In contrast to traditional parser generators like YACC or ANTLR, jparsec grammar is written in native Java language and is defined in terms of special Java instances of a `Parser` class. Each parser object represents a grammar rule and can be combined with other parser objects to create more complex production rules. For example, a production rule of the form "A ::= $B \mid C \mid D$" can be created by the following Java code:

```
Parser<Foo> a = Parsers.or(b, c, d);
```

where `b`, `c`, and `d` are parser instances representing the non-terminals B, C, and D in our production rule. In jparsec, once a parser object is created, it can be asked to "parse" a given input:

```
a.parse("text to be parsed");
```

Depending on how the parsers are defined, the return value (the result of parsing) can be a value resulting from a calculation or an abstract syntax tree representing the input text.

This feature of jparsec appeared to be very beneficial for CoreASM. Upon loading a specification, the kernel provides references to the core parser objects (such as white spaces, identifiers, terms, etc.)[6] and make them available for plugins to build upon.

[4] http://www.cs.rit.edu/~ats/

[5] http://jparsec.codehaus.org

[6] Some of these core parsers, such as the one for parsing CoreASM terms, can also be extended by plugins.

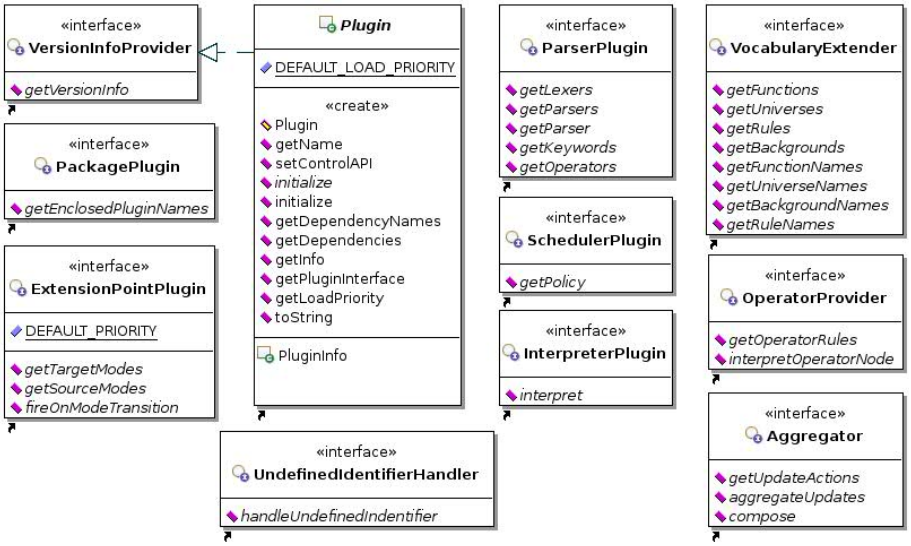


Figure 6.4: CoreASM Plugin Interfaces

Plugins in turn provide their contributions to the parser in form of new jparsec parser objects. The kernel then puts all these contributions together to create the final parser that will be used to parse the specification.

6.2.2 CoreASM Plugins

Every CoreASM plugin must extend the abstract class `Plugin` and most likely implements at least one of the nine plugin interfaces offered by the engine (see Figure 6.4).[7] We introduced the seven most important plugin interfaces in Section 4.5; the remaining two are the `PackagePlugin` and the `UndefinedIdentifierHandler` interface. The former should be implemented by plugins that are defined to serve as a \package" of other plugins. For example, CoreASM comes with a *Standard Plugin* which is a plugin that implements only the `PackagePlugin` interface and when loaded (see LoadSpecPlugins on page 34) provides a list of plugins that it consists of. The latter one, `UndefinedIdentifierHandler`, is implemented by plugins that offer a mechanism to deal with undefined identifiers. For example, a plugin can implement this interface and override the default behavior of the engine and generate an error whenever an undefined identifier is recognized by the engine; see Section 4.2.2 and the definition of the rule HandleUndefinedIdentifier in Section A.2.

A CoreASM plugin is most likely accompanied by a number of auxiliary Java classes. As a result, every CoreASM plugin is expected to be packed into a single JAR

[7] Even if a plugin does not implement any of the plugin interfaces, it is still a valid plugin as long as it properly extends the `Plugin` class. However, the effect of loading such a plugin would be extremely limited.

file[8] together with an identification file. When an instance of `Engine` is initialized, it searches a specific *plugin* folder, creates a catalog of available plugins (abstractly modeled by the LoadCatalog rule on page 32) and loads the plugin class files together with their corresponding classes into the Java Virtual Machine (JVM), so that they can be later instantiated if needed. As a result, to add a new plugin to CoreASM, one only needs to put the JAR file of the compiled plugin into the *plugin* folder of the engine.

6.3 User Interfaces and Tools

The CoreASM engine is implemented as a Java component and requires a *driver* program (such as a user interface) to run the engine, e.g., to pass specification files to the engine and to control its simulation run by manipulating parameters. There are currently two user interfaces available for the CoreASM engine: a powerful command-line tool called Carma, and a graphical interactive development environment in the Eclipse platform, known as the CoreASM Eclipse Plugin.

Carma

Carma is a comprehensive command-line user interface for CoreASM that offers rich control over the runs of the engine through more than a dozen command-line options and switches. To execute a specification, users can simply run Carma on the command line and pass it the name of the specification file as an argument. By default, Carma does not have a termination condition, but it offers a number of termination conditions, such as termination after a number of steps, termination on empty updates, and termination when there is no valid agent with a defined program. As an example, the following command runs the CoreASM specification `MySpec.coreasm` using Carma and stops after 30 steps or after a step that generates empty updates; it also provides a print-out of the final state before termination.

```
carma --steps 30 --empty-updates --dump-final-state MySpec.coreasm
```

The **CoreASM** Eclipse Plugin

The CoreASM Eclipse Plugin is a graphical interactive development environment for CoreASM in form of a plugin for the well-known Eclipse software development platform. The IDE provides various options to control execution of CoreASM specifications. The plugin extends the Eclipse platform to support dynamic syntax highlighting and interactive execution of CoreASM specifications. Since the language of CoreASM for a given specification is defined by the set of plugins used by that specification, with every change to the specification, the editor component of the CoreASM

[8] JAR (Java Archive) files are package files that are used by software developers to distribute Java classes and their associated metadata.

Eclipse Plugin passes the specification to the CoreASM engine and gets the set of plugins that are used by the specification. The editor then asks the plugins for the set of keywords, functions, universes and backgrounds they provide and uses this information to offer a dynamic syntax highlighting of the specification.

Figure 6.5(a) shows a snapshot of the CoreASM environment in Eclipse. At the top left corner (1), the toolbar is extended to include buttons to pause, resume and stop a simulation run. The editor (2) provides dynamic syntax highlighting for CoreASM specifications based on the set of CoreASM plugins used in the specification. A configurable output console (3) provides a print-out of the results of the simulation with optional additional information on the simulation process and the state of the simulated machine.

6.3.1 CSDe

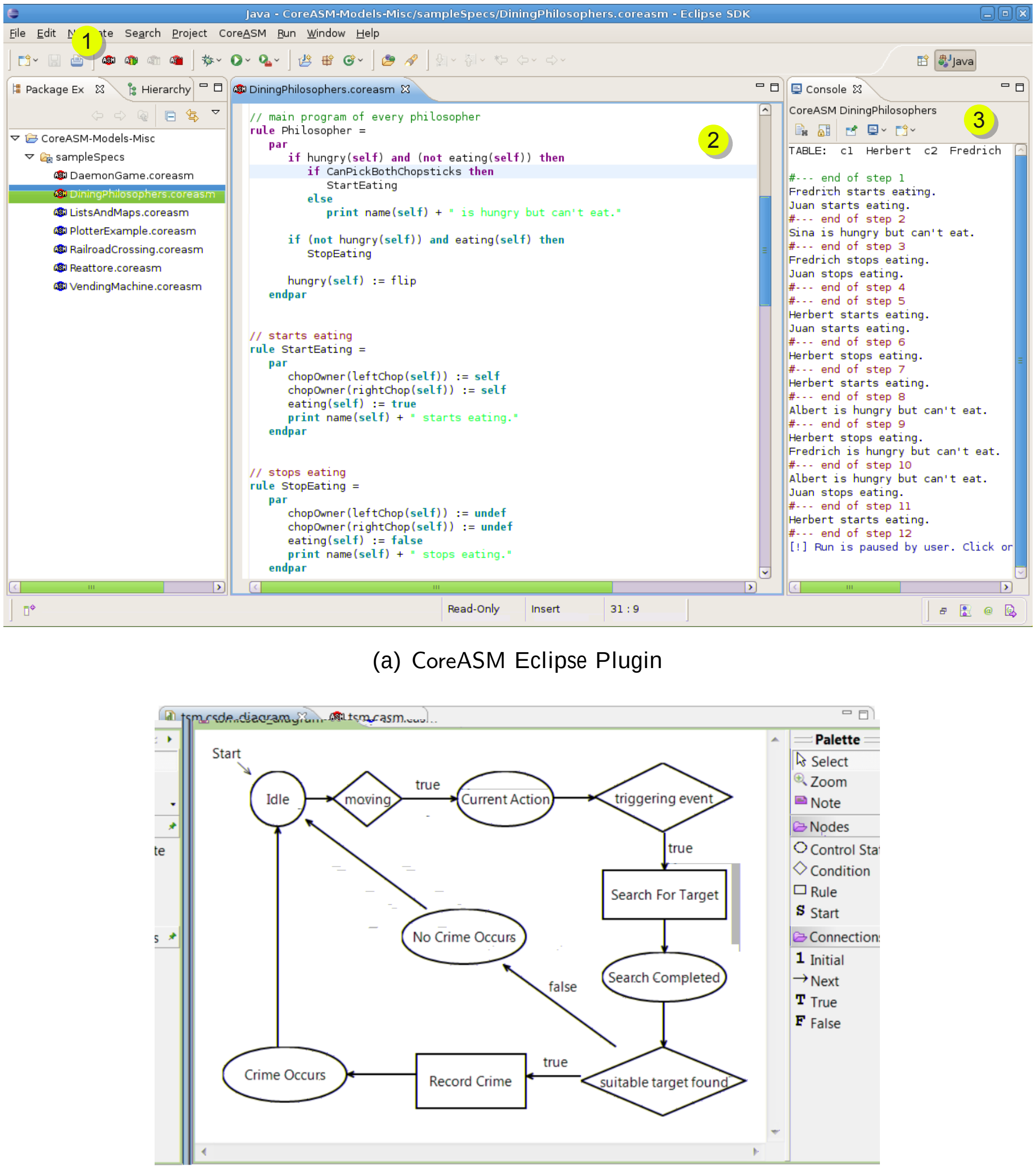
The Control State Diagram editor (CSDe), under development by Piper J. Jackson [31], is a sophisticated tool for creating and modifying Control State ASMs and translating them into CoreASM specifications. The tool is implemented as a plugin for the Eclipse software development platform. The plugin allows the user to work with Control State Diagrams (CSDs) using a point-and-click schema (see Figure 6.5(b)).

The simplicity of control state diagrams and the intuitiveness of the graphical user interface work together to allow users to confidently contribute to the design, regardless of their technical background. The diagram editor (CSDe) is capable of automatically transforming diagrams into CoreASM specifications. Since control state diagrams do not necessarily include initial states of the system or other more concrete information required for machine execution, such specifications may not be directly executable. However, they provide an abstract structure for the design of systems and act as foundations for further development of the specifications. The automatic translation feature facilitates the transition from high-level design ideas expressed in graphical form towards abstract yet relatively more concrete specifications.

6.3.2 Model Checking **CoreASM** Specifications

The CoreASM engine facilitates experimental validation of ASM models by providing the means to execute abstract specifications and to explore behavioral aspects in an interactive fashion. However, experimental validation without model checking cannot formally verify the correctness of a system with respect to all of its possible behaviors. In order to provide model checking support for CoreASM, George Ma developed a tool called CoreASM2Promela [62] that utilizes the CoreASM engine to translate CoreASM models into equivalent Promela models which can be verified using the Spin model checker.[9] From a high level perspective, the steps in the translation and verification process are as follows: (i) a CoreASM specification is loaded and

[9] Spin is a widely used automata based model checker that has been used extensively in the design of asynchronous distributed systems [53].


Java - CoreASM-Models-Misc/sampleSpecs/DiningPhilosophers.coreasm - Eclipse SDK
File Edit Navigate Search Project CoreASM Run Window Help
1
Java
Package Ex
Hierarchy
CoreASM-Models-Misc
sampleSpecs
DaemonGame.coreasm
DiningPhilosophers.coreasm
ListsAndMaps.coreasm
PlotterExample.coreasm
RailroadCrossing.coreasm
Reattore.coreasm
VendingMachine.coreasm
DiningPhilosophers.coreasm
2
Console
CoreASM DiningPhilosophers
3
Read-Only
Insert
31 : 9

```
// main program of every philosopher
rule Philosopher =
   par
      if hungry(self) and (not eating(self)) then
         if CanPickBothChopsticks then
            StartEating
         else
            print name(self) + " is hungry but can't eat."

      if (not hungry(self)) and eating(self) then
         StopEating

      hungry(self) := flip
   endpar


// starts eating
rule StartEating =
   par
      chopOwner(leftChop(self)) := self
      chopOwner(rightChop(self)) := self
      eating(self) := true
      print name(self) + " starts eating."
   endpar


// stops eating
rule StopEating =
   par
      chopOwner(leftChop(self)) := undef
      chopOwner(rightChop(self)) := undef
      eating(self) := false
      print name(self) + " stops eating."
   endpar
```

```
TABLE:  c1  Herbert  c2  Fredrich

#--- end of step 1
Fredrich starts eating.
Juan starts eating.
#--- end of step 2
Sina is hungry but can't eat.
#--- end of step 3
Fredrich stops eating.
Juan stops eating.
#--- end of step 4
#--- end of step 5
Herbert starts eating.
Juan starts eating.
#--- end of step 6
Herbert stops eating.
#--- end of step 7
Herbert starts eating.
#--- end of step 8
Albert is hungry but can't eat.
#--- end of step 9
Herbert stops eating.
Fredrich is hungry but can't eat.
#--- end of step 10
Albert is hungry but can't eat.
Juan stops eating.
#--- end of step 11
Herbert starts eating.
#--- end of step 12
[!] Run is paused by user. Click or
```

(a) CoreASM Eclipse Plugin

(b) CSDe: A Control State Diagram editor for CoreASM

Figure 6.5: CoreASM Tools in Eclipse

parsed by the CoreASM engine, producing an abstract syntax tree; (ii) the tree is translated into Promela; (iii) Spin is invoked to generate a verifier of the Promela model, producing C code; (iv) the C code is compiled, generating a custom verifier of the CoreASM specification; (v) the verifier is run, producing a counter example if the property being checked does not hold.

In order to properly translate CoreASM specifications into Promela models, we needed to extend the CoreASM language by two new plugins, namely the *Signature Plugin* (see Section 5.4.1) and the *Property Plugin*, to support declaration of function signatures and specification of LTL properties as part of CoreASM specifications. The Property Plugin is a small plugin that allows correctness properties, 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 plugin to check simple global assertions). Correctness properties are only applicable during model checking, and are translated by our CoreASM to Promela translator.

The Property plugin provides the following pattern to declare new LTL properties:

[check] property *LTL-property*

Including the keyword **check** with a property declaration indicates that the property should be checked during model checking.

Since Spin does not allow LTL properties to be included directly in a specification, the Property plugin was developed to improve the usability of the model checker. 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 plugin allows multiple properties to be specified for a single specification.

George Ma has successfully used CoreASM2Promela to model check several nontrivial ASM specifications; the details of the case studies and a comprehensive discussion of the results are presented in George Ma's M.Sc thesis [62]. However, there are certain limitations in model checking abstract ASM specifications using Spin. For example, as Spin can only check finite models, the translation scheme is limited to CoreASM specifications which have finite states. Thus, the translation supports only static universes and enumerated backgrounds.

Chapter 7

Conclusions and Perspectives

This work presented the design and development of the CoreASM modeling framework and tool environment for high-level design and analysis of abstract state machine models. The CoreASM engine forms the kernel of a novel environment for model-based engineering of abstract operational requirements and design specifications at the early phases of the software design and development process. Focusing on *freedom of experimentation* and *design exploration*, CoreASM offers a flexible modeling environment that facilitates writing of easily modifiable, concise and understandable formal specifications by minimizing the need for encoding of domain concepts into the constructs of the language.

In order to minimize the cost of such encoding, the CoreASM language and tool architecture are both designed to be easily extensible so that they can be customized for specific application contexts, thus realizing *domain-specific* ASM dialects. The ASM literature contains many examples of using such ASM dialects: many published specifications of large systems have introduced background elements or non-standard rule forms that were well suited to express the intended behavior at an appropriate level of abstraction in the given domain. By similarly allowing the customization of the CoreASM language, we provide the benefits of executable specifications without loosing the expressiveness of a domain-specific language, and avoid the introduction of a further encoding level between the conceptual specification and its executable version.

The design of the CoreASM engine is formally specified in ASMs. The entire lifecycle of the CoreASM engine is defined as an extensible control-state ASM and the CoreASM language is formally defined through the specification of an interpreter (in the form of an abstract state machine) that ensures the executability of the language and provides its formal semantics.

CoreASM has been recognized by the ASM community and has been used by various research groups in Europe, Asia, and North America [61, 1, 4, 56, 63, 23].[1] Based

[1] To name a few, CoreASM has been applied in a number of research projects at the Computer Science Department of the University of Pisa in Italy, the Embedded Software Laboratory at the

on solid experience gained through the practical use of CoreASM in a number of diverse application domains (see Chapter ??), we claim that CoreASM serves practical needs of high-level modeling and rapid prototyping of complex distributed systems and will be an asset for industrial engineering of complex software systems by making software specifications and designs more robust and reliable. Prior to actually building a system, CoreASM specifications facilitate development of concise blueprints for intuitive reasoning about key system attributes, supporting requirements specification, design analysis, validation and (where appropriate) formal verification of system properties.

7.1 Significance of the Contribution

Among all the existing ASM tool environments, CoreASM stands out as being the closest to the spirit of abstract state machines [20]. Here, we summarize the most significant features that distinguish it from other ASM tools.

A Rich ASM Language and Framework

CoreASM offers a rich ASM language with a syntax that closely follows the pseudocode style of ASMs and a formally defined semantics that is faithful to the original ASM semantics as defined in [20]. CoreASM is the first ASM tool environment that directly supports distributed ASM computation models with custom scheduling policies. Its language supports classes of basic, distributed, and Turbo ASMs, making it the most comprehensive ASM language available.

Encouraging Rapid Prototyping

The CoreASM language is an untyped language with a minimal yet human-readable syntax that facilitates writing abstract and untyped models which can be refined into more concrete versions as needed. Thus, it encourages rapid prototyping of abstract machine models for testing and design space exploration, and facilitates agile software development [42]. An independent study performed by Jensen et al. [56], comparing the abstraction level of specifications written in CoreASM and AsmL[2], shows that the CoreASM language can be used to specify algorithms in a higher level of abstraction compared to AsmL. In their example of a data clustering algorithm, the CoreASM description of the algorithm is 82 lines, almost half the size of the 155 lines of AsmL description of the same algorithm (see [56, Fig. 4]). The authors conclude that compared to AsmL, CoreASM is more suited for the early stages of software engineering.

RWTH Aachen University in Germany, the Open Systems Development Group at the University of Agder in Norway, and the Department of Computer Science and Engineering at the Anna University in India.

[2] The executable ASM language developed by the Foundation of Software Engineering group at Microsoft [66]

Extensible Language and Architecture

The most significant feature of CoreASM is the extensibility of its language and modeling environment. To reduce the cost of writing specifications, one has to minimize the need for encoding in mapping the problem space to a formal model. This approach usually leads to the design of domain-specific languages. The CoreASM extensibility framework provides utmost flexibility for extending its language definition and execution engine in order to tailor it to the particular needs of virtually any conceivable application context. This allows CoreASM to be used very much in the same way ASMs were meant to be used.

An Open Framework

CoreASM is one of the few ASM tools that is implemented as an open framework. Developed in Java | a platform independent, open source programming language | and under an open source license, CoreASM can be modified, extended and improved as needed by its user community. The CoreASM engine comes with a simple yet comprehensive API that offers full access to the states of simulated machines and complete control over the execution of CoreASM specifications, and as such facilitates the integration of CoreASM as an ASM simulator component into other applications.

7.2 Future Work

The CoreASM project is in continuous development. Currently, the execution engine can execute standard ASM specifications; various plugins offer common backgrounds such as numbers, sets, strings, and lists, and more specialized plugins offer sophisticated features such as the JASMine plugin for interfacing ASM specifications with Java class libraries (see Section 5.5).

However, there are a number of open issues that have not been yet sufficiently addressed by the CoreASM project. In this section we review some of these issues and discuss them as possible subjects of future work.

Debugging Features

Traditional debugging models of programming (e.g. step by step execution of instructions) do not suit ASMs. There is no such concept as the \current" instruction, nor an explicit notion of \stepping" over instructions. However, similar notions can be applied to computation steps of ASMs instead.

For example, a debugging user interface can offer, after every step of simulation, the option of browsing the ASM program as a tree of rule constructs annotated with the most recently generated update multisets produced by the rules. Such a feature would allow users to investigate the changes (updates) produced by different parts of ASM programs at desired levels of detail.

The CoreASM engine provides the necessary services (such as step-by-step execution of the engine, full access to the simulated state, and the possibility of applications to intervene in the execution process of the engine) supporting the implementation of various debugging features by a CoreASM user interface. Non-trivial debugging features, however, are not yet implemented in any of the currently available CoreASM user interfaces.

Type System

The CoreASM language is designed as a primarily typeless language to encourage rapid prototyping of abstract specifications. Although dynamic types are attached to every CoreASM value (element) and various primitive and complex data types are provided by plugins (see sections 5.2 and 5.3), there is no concept of static typing or a type system defined in the kernel of CoreASM. State locations (a more generalized notion of programming variables) are essentially typeless and there is no type-checking offered by the CoreASM kernel.

The Signature plugin (see Section 5.4.1) extends the CoreASM language and the engine by offering a means to define type signatures for state locations. It also provides runtime type checking on function calls and on updates to locations for which a signature is defined. However, much more can be done in this domain. For example, collection plugins could be improved to offer parameterized type constructors and the Signature plugin could be extended to offer static type analysis of fully-typed specifications, a practical requirement for model checking of CoreASM specifications.

Literate Specifications

Following the idea of *literate specifications* [57] (an extension of Knuth's *literate programming technique* [59]), it would be beneficial to integrate facilities for writing CoreASM specifications into various document preparation systems such as OpenOffice Writer[3] or the LaTeX typesetting system[4]. Such an integration would facilitate the development of compound system documents, consisting of executable specifications and system documentations, that not only provide formal specification of systems, but also offer design rationale and necessary explanation on how such systems work.

The current implementation of CoreASM can import specifications from OpenOffice Writer documents and the Carma user interface (see Section 6.3) can load and execute OpenOffice Writer documents containing CoreASM fragments. The CoreASM engine could be extended to also support import and export of specifications to and from LaTeX documents.[5]

[3] http://www.openoffice.org/

[4] http://www.latex-project.org/

[5] A basic CoreASM-to-LaTeX export feature has already been implemented in Carma which has been used to produced the color-annotated specification of Appendix B.1.

Integrated Development Environment

The CoreASM IDE, a combination of the CoreASM Eclipse plugin and the CSD editor (see Section 6.3), is still in early development. We envision further improvements providing debugging features (discussed above) and enhanced coding assistance features, such as easy navigation between different layers of abstraction and refinements, which would be of real value in building complex models.

Verification and Model Checking

A proper formal specification facilitates establishing the validity of the initial formalization step, which itself is a prerequisite for any meaningful approach to formal verification. However, the only machine-assisted verification supported by the current implementation of CoreASM is in the form of rudimentary model checking (see [62] and Section 6.3.2). More sophisticated interfaces to existing model checking tools are needed to fully exploit the potential they provide.

Automatic Code and Test Case Generation

There is currently no support for automatic code generation from CoreASM models. The CoreASM engine is reasonably fast and efficient for interactive modeling and experimental validation; nonetheless, there is room for improving performance by generating Java or C++ code from CoreASM specifications. Automatic test case generation for conformance testing, comparable to AsmL Spec Explorer [73], is a work in progress independent of our work.

Appendices

Appendix A

Supplementary Definitions

A.1 Abstract Storage

- PushState puts the current state in the stack. We assume that $stack_{state}$ is empty in the initial state.

```
**PushState** $\equiv$
  Push($stack_{state}, state$)
```

- PopState retrieves the state from the top of the stack, thus discarding the current state.

```
**PopState** $\equiv$
  $state := top(stack_{state})$
  Pop($stack_{state}$)
```

- Apply(u) applies the updates in the update set u to the current state.

```
**Apply**($u$) $\equiv$
  **forall** $(l, v) \in u$ **do**
    SetValue($l, v$)
```

- ClearState resets $state$ to an empty state.

```
**ClearState** $\equiv$
  **let** $s = new(\textsc{State})$ **in**
    $state := s$
```

- $newElement$: $\textsc{Element}$
  returns a new element; i.e., imports a new element into the state and returns the imported element. This function is defined as follows:

$$newElement \equiv new(\textsc{Element})$$

- $\mathit{inconsistentUpdates} : \textsc{Set}(\textsc{Update}) \mapsto \textsc{Set}(\textsc{Update})$
  returns the set of inconsistent updates (according to [20, Def. 2.4.5]) in the given update set. We assume that the update set consists of regular updates only (i.e. actions are $\mathit{updateAction}$).

  $$\mathit{inconsistentUpdates}(\mathit{uset}) \equiv \{(l, v, a) \in \mathit{uset} \mid \exists(l', v', a') \in \mathit{uset},\ l = l' \wedge v \neq v'\}$$

- $\mathit{isConsistent} : \textsc{Set}(\textsc{Update}) \mapsto \textsc{Boolean}$
  returns true if the update set is consistent according to [20, Def. 2.4.5]. We assume that the update set consists of regular updates only (i.e. actions are $\mathit{updateAction}$).

  $$\mathit{isConsistent}(\mathit{uset}) \equiv |\mathit{inconsistentUpdates}(\mathit{uset})| > 0$$

- $\mathit{isUniverseName} : \textsc{Name} \mapsto \textsc{Boolean}$

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

- $\mathit{isFunctionName} : \textsc{Name} \mapsto \textsc{Boolean}$

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

- $\mathit{isRuleName} : \textsc{Name} \mapsto \textsc{Boolean}$

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

A.2 Interpreter

- ClearTree(t) clears the given tree from any assigned value, location, or updates.

```
**ClearTree**($\alpha$) $\equiv$
  **if** $\alpha \neq \mathit{undef}$ **then**
    $\mathit{value}(\alpha) := \mathit{undef}$
    $\mathit{update}(\alpha) := \mathit{undef}$
    $\mathit{loc}(\alpha) := \mathit{undef}$
    ClearTree($\mathit{first}(\alpha)$)
    ClearTree($\mathit{next}(\alpha)$)
```

- CopyTree($t, \mathit{setNext}$) creates a copy of the given tree, without copying assigned values, locations, or updates. If $\mathit{setNext}$ is true, it also copies the next sibling of the given root node.

```
CopyTree($\alpha, setNext$) $\equiv$
  if $\alpha \neq undef$ then
    let $n$ = $new$(NODE) in
      $class(n)$ := $class(\alpha)$
      $pattern(n)$ := $pattern(\alpha)$
      $token(n)$ := $token(\alpha)$
      $grammarRule(n)$ := $grammarRule(\alpha)$
      $plugin(n)$ := $plugin(\alpha)$
      $first(n)$ := CopyTree($first(\alpha), true$)
      if $setNext$ then
        $next(n)$ := CopyTree($next(\alpha), true$)
      result := $n$
  else
    result := $undef$
```

- CopyTreeSub($\alpha, \langle x_1, \ldots, x_n \rangle, \langle \lambda_1, \ldots, \lambda_n \rangle$) returns a copy of the given parse tree α, where every instance of a parameter node x_i is substituted by a copy of the corresponding argument λ_i. We assume that the elements in the formal parameters list (x_i's) are all distinct. Also, formal parameters substitution is applied only to occurrences of formal parameters in the original tree passed as argument, and *not* also on the actual parameters themselves.

```
CopyTreeSub($\alpha, \langle x_1, \ldots, x_n \rangle, \langle \lambda_1, \ldots, \lambda_n \rangle$) $\equiv$
  if $\alpha \neq undef$ then
    if $class(\alpha)$ = Id $\wedge \exists i$ s.t. $token(\alpha) = x_i$ then
      $result \leftarrow$ CopyTree($\lambda_i, false$)
    else
      let $n$ = $new$(NODE) in
        $first(n) \leftarrow$ CopyTreeSub($first(\alpha), \langle x_1, \ldots, x_n \rangle, \langle \lambda_1, \ldots, \lambda_n \rangle$)
        $next(n) \leftarrow$ CopyTreeSub($next(\alpha), \langle x_1, \ldots, x_n \rangle, \langle \lambda_1, \ldots, \lambda_n \rangle$)
        $class(n)$ := $class(\alpha)$
        $pattern(n)$ := $pattern(\alpha)$
        $token(n)$ := $token(\alpha)$
        $grammarRule(n)$ := $grammarRule(\alpha)$
        $plugin(n)$ := $plugin(\alpha)$
        result := $n$
  else
    result := $undef$
```

- HandleUndefinedIdentifier($pos, x, args$) asks all the plugins registered to handle undefined identifiers to evaluate the node with the undefined identifier (pos). It is considered an error if more than one plugin evaluates the undefined identifier with different results. If none of the plugins could evaluate the node, KernelHandleUndefIdentifier will be called to create a new function element with a default value of undef$_e$ for the given arguments.

```
HandleUndefinedIdentifier($pos, x, args$) $\equiv$
  local $results$ [$results$ := {}] in
    seq
      foreach $p$ in $loadedPlugins$ do
        seqblock
          ClearTree($pos$)
          PluginHandleUndefIndentifier($p, pos, x, args$)
          if $evaluated(pos)$ then
            add $\langle p, loc(pos), updates(pos), value(pos)\rangle$ to $results$
        endseqblock

    next
      if $|results|$ = 0 then
        KernelHandleUndefIdentifier($pos, x, args$)
      else
        choose $\langle p, l, u, v\rangle$ in $results$ with $\exists\langle p', l', u', v'\rangle \in results, \langle l, v, u\rangle \not\equiv \langle l', v', u'\rangle$ do
          Error(`There is an ambiguity in resolving the identifier.')
        ifnone
          $[\![pos]\!]$ := $(l, u, v)$
```

A.3 Scheduler

- $updateInstructions$: Multiset(Update)
  is the multiset of accumulated update instructions in the current computation step.

- $updateSet$: Set(Update)
  is the set of (aggregated) updates in last computation step.

- $selectedAgentsSet$: Set(Element)
  is the set of selected agents contributing to the computation of the current step.

- $initAgent$: Element
  is the initial agent the engine creates to run the $init$ rule.

- $chosenAgent$: Element
  is the currently running (or to be running) agent.

- $chosenProgram$: Rule
  is the rule element that represents the program of the chosen agent. The value of this function is set by the Abstract Storage.

- $morePossibleSetsExist$: Boolean
  holds true if there are more possible combinations of agents that can contribute to the current computation step.

- $\mathit{isSingleAgentInconsistent} : \textsc{Boolean}$
  holds true if the last inconsistent set of updates is produced by a single agent.

$$\begin{aligned}&\mathit{isSingleAgentInconsistent} \equiv\\ &\quad \exists a \in \textsc{Element}, \exists l \in \textsc{Location}, \forall u_1, u_2 \in \mathit{updateSet},\\ &\qquad \mathit{uiLoc}(u_1) = \mathit{uiLoc}(u_2) \wedge \mathit{uiAgents}(u_1) = \mathit{uiAgents}(u_2) = \{a\}\end{aligned}$$

- LoadSchedulingPolicy, based on the set of loaded plugins, loads a scheduling policy for scheduling of agents in every computation step.

```
LoadSchedulingPolicy ≡
  let $policies = \{pluginSchedulingPolicy(p) \mid p \in specPlugins \wedge isPolicyPlugin(p)\} \setminus \{undef\}$ in
    if $|policies| = 0$ then
      $schedulingPolicy := undef$
    else
      if $|policies| = 1$ then
        choose $policy \in policies$ do
          $schedulingPolicy := policy$
          $schedulingGroup := newSchedulingGroup(policy)$
      else
        Error(`Conflicting scheduling policies.')
```

A.4 Control API

The following functions and rules define the interface of the engine to its environment.

- $\mathit{specification} : \textsc{Spec}$
  is the current CoreASM specification loaded by the engine.

- $\mathit{pluginCatalog} : \textsc{Set}(\textsc{Plugin})$
  is the set of all the plugins available to the engine.

- $\mathit{loadedPlugins} : \textsc{Set}(\textsc{Plugin})$
  is the set of loaded plugins by the engine.

- $\mathit{grammarRules} : \textsc{Set}(\textsc{GrammarRule})$
  is the set of all the grammar rules provided by the kernel and loaded plugins.

- $\mathit{isStateInitialized} : \textsc{Boolean}$
  holds true if the simulation state is initialized.

- $\mathit{stepCount} : \textsc{Number}$
  is the simulation step counter.

- $\mathit{state} : \textsc{State}$
  holds the current simulation state.

- $\mathit{agentSet} : \textsc{Set}(\textsc{Element})$
  is the set of all the available agents in the current state retrieved from the Abstract Storage at the beginning of every computation step.

- $\mathit{engineProperties} : \textsc{Name} \mapsto \textsc{Name}$
  holds all the defined engine properties and their values. The behavior of the engine (and its plugins) can be customized by these properties.

- $\mathit{engineMode} : \textsc{EngineMode}$
  returns the current execution mode of the engine.

- $\mathit{isEngineBusy} : \textsc{Boolean}$

$$\mathit{isEngineBusy} \equiv \mathit{engineMode} \not\in \{\mathit{Idle}, \mathit{Error}\}$$

- UpdateState($\mathit{updates}$), if $\neg\mathit{isEngineBusy}$, updates the current state by applying the given set of updates.

- Step puts a step command in the command queue of the engine.

A.5 Plugins

A.5.1 Choose Rule Plugin

Choose Rule

```
$\llparenthesis$ choose $^{\alpha}x$ in $^{\beta}\boxed{e}$ do$^{\gamma}\boxed{r}$ $\rrparenthesis$   $\rightarrow$   $pos$ := $\beta$
$\llparenthesis$ choose $^{\alpha}x$ in $^{\beta}v$ do$^{\gamma}\boxed{r}$ $\rrparenthesis$   $\rightarrow$   if $enumerable(v)$ then
                                                   let $s$ = $enumerate(v)$ in
                                                     if $|s| > 0$ then
                                                       choose $t \in s$ do
                                                         AddEnv($x, t$)
                                                       $pos$ := $\gamma$
                                                     else
                                                       $[\![pos]\!]$ := $(undef, \{\!\!\}, undef)$
                                                 else
                                                   Error(`Cannot choose from a non-enumerable element.')
$\llparenthesis$ choose $^{\alpha}x$ in $^{\beta}v$ do$^{\gamma}u$ $\rrparenthesis$   $\rightarrow$   RemoveEnv($x$)
                                                 $[\![pos]\!]$ := $(undef, u, undef)$
```

Choose Rule

```
$\llparenthesis \textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}\boxed{e}_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis \rightarrow$
                                        $\mathit{pos} := \beta$
                                        $\mathit{considered}(\beta) := \{\}$
$\llparenthesis \textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis \rightarrow$
                                        $\textbf{if}\ \mathit{enumerable}(v_1)\ \textbf{then}$
                                          $\textbf{let}\ s = \mathit{enumerate}(v_1) \setminus \mathit{considered}(\beta)\ \textbf{in}$
                                            $\textbf{if}\ |s| > 0\ \textbf{then}$
                                              $\textbf{choose}\ t \in s\ \textbf{do}$
                                                $\mathsf{AddEnv}(x, t)$
                                                $\mathit{considered}(\beta) := \mathit{considered}(\beta) \cup \{t\}$
                                              $\mathit{pos} := \gamma$
                                            $\textbf{else}$
                                              $[\![\mathit{pos}]\!] := (\mathit{undef}, \{\!|\!\}, \mathit{undef})$
                                        $\textbf{else}$
                                          Error(`Cannot choose from non-enumerable element')

$\llparenthesis \textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}\boxed{r}\ \rrparenthesis \rightarrow$   $\textbf{if}\ v_2 = \mathsf{true}_e\ \textbf{then}$
                                                   $\mathit{pos} := \delta$
                                                 $\textbf{else}$
                                                   $\mathit{pos} := \beta$
                                                   $\mathsf{RemoveEnv}(x)$
                                                   $\mathsf{ClearTree}(\gamma)$
$\llparenthesis \textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}u \rrparenthesis \rightarrow$   $\mathsf{RemoveEnv}(x)$
                                                 $[\![\mathit{pos}]\!] := (\mathit{undef}, u, \mathit{undef})$
```

Choose Rule

```
$(\!|\,\textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}\boxed{e}_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \ \textbf{ifnone}\ {}^{\epsilon}\boxed{r}\ |\!) \rightarrow$   $pos := \beta$
                                                                       $considered(\beta) := \{\}$
$(\!|\,\textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}\boxed{e}_2\ \textbf{do}^{\delta}\boxed{r}\ \ \textbf{ifnone}\ {}^{\epsilon}\boxed{r}\ |\!) \rightarrow$
    $\textbf{if}\ enumerable(v_1)\ \textbf{then}$
      $\textbf{let}\ s = enumerate(v_1) \setminus considered(\beta)\ \textbf{in}$
        $\textbf{if}\ |s| > 0\ \textbf{then}$
          $\textbf{choose}\ t \in s\ \textbf{do}$
            $\mathsf{AddEnv}(x, t)$
            $considered(\beta) := considered(\beta) \cup \{t\}$
          $pos := \gamma$
        $\textbf{else}$
          $pos := \epsilon$
    $\textbf{else}$
      $\mathsf{Error}(\text{`Cannot choose from non-enumerable element'})$
$(\!|\,\textbf{choose}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}\boxed{r}\ \ \textbf{ifnone}\ {}^{\epsilon}\boxed{r}\ |\!) \rightarrow$   $\textbf{if}\ v_2 = \mathsf{true}_e\ \textbf{then}$
                                                                      $pos := \delta$
                                                                    $\textbf{else}$
                                                                      $pos := \beta$
                                                                      $\mathsf{RemoveEnv}(x)$
                                                                      $\mathsf{ClearTree}(\gamma)$
$(\!|\,\textbf{choose}\ {}^{\alpha}x \in {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}v_2\ \textbf{do}^{\delta}u\ \textbf{ifnone}\ {}^{\epsilon}\boxed{r}\ |\!) \rightarrow$   $\mathsf{RemoveEnv}(x)$
                                                                  $[\![pos]\!] := (undef, u, undef)$
$(\!|\,\textbf{choose}\ {}^{\alpha}x \in {}^{\beta}v_1\ \textbf{with}\ {}^{\gamma}e_2\ \textbf{do}^{\delta}\boxed{r}\ \ \textbf{ifnone}\ {}^{\epsilon}u\,|\!) \rightarrow$   $[\![pos]\!] := (undef, u, undef)$
```

A.5.2 Forall Rule Plugin

Forall Rule

```
$(\!|\,\textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}\boxed{e}\ \ \textbf{do}^{\gamma}\boxed{r}\ |\!)$   $\rightarrow$   $pos := \beta$
                                              $[\![pos]\!] := (undef, \{\!|\,|\!\}, undef)$
                                              $considered(\beta) := \{\}$
$(\!|\,\textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v\ \textbf{do}^{\gamma}\boxed{r}\ |\!)$     $\rightarrow$   $\textbf{if}\ enumerable(v)\ \textbf{then}$
                                                $\textbf{let}\ s = enumerate(v) \setminus considered(\beta)\ \textbf{in}$
                                                  $\textbf{if}\ |s| > 0\ \textbf{then}$
                                                    $\textbf{choose}\ t \in s\ \textbf{do}$
                                                      $\mathsf{AddEnv}(x, t)$
                                                      $considered(\beta) := considered(\beta) \cup \{t\}$
                                                    $pos := \gamma$
                                              $\textbf{else}$
                                                $\mathsf{Error}(\text{`Cannot enumerate a non-enumerable element'})$
$(\!|\,\textbf{forall}\ {}^{\alpha}x\ \textbf{in}\ {}^{\beta}v\ \textbf{do}^{\gamma}u\,|\!)$        $\rightarrow$   $pos := \beta$
                                              $\mathsf{RemoveEnv}(x)$
                                              $\mathsf{ClearTree}(\gamma)$
                                              $[\![pos]\!] := (undef, updates(pos) \cup u, undef)$
```

A.5.3 Predicate Logic Plugin

The *and* Operator

Predicate Logic Plugin: and

```
$(\!| {}^{\alpha}\boxed{?}$ and ${}^{\beta}\boxed{?} |\!)_{[400]}$ $\rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                          $pos := \lambda$
                        ifnone
                          if $isBoolean(value(\alpha)) \wedge isBoolean(value(\beta))$ then
                            if $(value(\alpha) = \mathsf{true}_e) \wedge (value(\beta) = \mathsf{true}_e)$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                            else
                              $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
```

The *or* Operator

Predicate Logic Plugin: or

```
$(\!| {}^{\alpha}\boxed{?}$ or ${}^{\beta}\boxed{?} |\!)_{[350]}$ $\rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                          $pos := \lambda$
                        ifnone
                          if $isBoolean(value(\alpha)) \wedge isBoolean(value(\beta))$ then
                            if $(value(\alpha) = \mathsf{true}_e) \vee (value(\beta) = \mathsf{true}_e)$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                            else
                              $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
```

The *xor* Operator

Predicate Logic Plugin: xor

```
$(\!| {}^{\alpha}\boxed{?}$ xor ${}^{\beta}\boxed{?} |\!)_{[350]}$ $\rightarrow$  choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                          $pos := \lambda$
                        ifnone
                          if $isBoolean(value(\alpha)) \wedge isBoolean(value(\beta))$ then
                            if $((value(\alpha) = \mathsf{true}_e) \vee (value(\beta) = \mathsf{true}_e)) \wedge$
                                  $((value(\alpha) = \mathsf{false}_e) \vee (value(\beta) = \mathsf{false}_e))$ then
                              $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                            else
                              $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
```

The *forall* Universal Quantifier

Predicate Logic Plugin: forall

```
$\llparenthesis \textbf{forall}^{\alpha} x \textbf{ in } {}^{\beta}\boxed{e} \textbf{ holds } {}^{\gamma}\boxed{e} \rrparenthesis$   →   $pos := \beta$
                                                                          $considered(\beta) := \{\}$
$\llparenthesis \textbf{forall}^{\alpha} x \textbf{ in } {}^{\beta}v \textbf{ holds } {}^{\gamma}\boxed{e} \rrparenthesis$   →   if $enumerable(v)$ then
                                                                            let $s = enumerate(v) \setminus considered(\beta)$ in
                                                                              if $|enumerate(v)| > 0$ then
                                                                                if $|s| > 0$ then
                                                                                  choose $t \in s$ do
                                                                                    AddEnv$(x, t)$
                                                                                    $considered(\beta) := considered(\beta) \cup \{t\}$
                                                                                  $pos := \gamma$
                                                                                else
                                                                                  $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                                                              else
                                                                                $[\![pos]\!] := (undef, undef, \mathsf{true}_e)$
                                                                          else
                                                                            Error(`Cannot enumerate a non-enumerable element')
$\llparenthesis \textbf{forall}^{\alpha} x \textbf{ in } {}^{\beta}v \textbf{ holds } {}^{\gamma}v \rrparenthesis$   →   if $(value(\gamma) = \mathsf{true}_e)$ then
                                                                            $pos := \beta$
                                                                          else
                                                                            $[\![pos]\!] := (undef, undef, \mathsf{false}_e)$
                                                                          RemoveEnv$(x)$
                                                                          ClearTree$(\gamma)$
```

A.5.4 Set Plugin

Set Comprehension Variant 2

Set Plugin : Set Comprehension variant 2

$\llparenthesis \{ \,{}^{\alpha}x \mid {}^{\beta_1}x_1 \ \mathbf{in}\ {}^{\gamma_1}\boxed{?}_1, \ldots, {}^{\beta}$

Set Plugin : Set Comprehension variant 3

```
$(\!|\{\ {}^{\alpha}x$ is ${}^{\epsilon}\boxed{e}\ \mid {}^{\beta_1}x_1$ in ${}^{\gamma_1}\boxed{?}_1, \ldots, {}^{\beta_n}x_n$ in ${}^{\gamma_n}\boxed{?}_n$ with ${}^{\delta}\boxed{?}\}|\!) \rightarrow$
        if $n \geq 1$ then
          if $\forall j \in [1..n],\ x \neq x_j$ then
            choose $j \in [1..n]$ with $value(\gamma_j) = undef$ do
              $pos := \gamma_j$
            ifnone
              if $sameNameTwoConstVar$ then
                Error(`No two constrainer variables may have the same name')
              else if $\exists c \in [1..n], \neg enumerable(value(\gamma_c))$ then
                Error(`Constrainer variables may only be bound to enumerable elements')
              else if $\exists c \in [1..n], |enumerate(value(\gamma_c))| = 0$ then
                $[\![pos]\!] := (undef, undef, newValue(setBack))$
              else
                $newSet(pos) := \{\}$
                InitializeChooseConsideredCombos
                $pos := \epsilon$
          else
            Error(`Constrainer variable cannot have same name as specifier')
        else
            Error(`At least one constrainer variable must be present')
where
  $sameNameTwoConstVar \equiv \exists k \in [1..n],\ \exists l \in [1..n]\ \ k \neq l \wedge x_k = x_l$

$(\!|\{\ {}^{\alpha}x$ is ${}^{\epsilon}\boxed{e}\ \mid {}^{\beta_1}x_1$ in ${}^{\gamma_1}v_1, \ldots, {}^{\beta_n}x_n$ in ${}^{\gamma_n}v_n$ with ${}^{\delta}v\}|\!) \rightarrow$
        if $value(\delta) := \mathrm{true}_e$ then
          $pos := \epsilon$
        else
          if OtherCombosToConsider then
            ChooseNextCombo
            ClearTree($\delta$)
            $pos := \delta$
          else
            DestroyConsideredCombos
          $[\![pos]\!] := (undef, undef, setElement(newSet(pos)))$
```

```
$\llparenthesis \{\ {}^{\alpha}x$ is ${}^{\epsilon}v \mid {}^{\beta_1}x_1$ in ${}^{\gamma_1}v_1, \ldots, {}^{\beta_n}x_n$ in ${}^{\gamma_n}v_n$ with ${}^{\delta}v\} \rrparenthesis \rightarrow$
        seq
          add $value(\epsilon)$ to $newSet(pos)$
        next
          if OtherCombosToConsider then
            ChooseNextCombo
            ClearTree($\delta$)
            ClearTree($\epsilon$)
            $pos := \delta$
          else
            DestroyConsideredCombos
            $[\![pos]\!] := (undef, undef, setElement(newSet(pos)))$
```

The Set Difference Operator

Set Plugin : difference

```
$\llparenthesis {}^{\alpha}\boxed{?} \backslash {}^{\beta}\boxed{?} \rrparenthesis_{[650]}$   $\rightarrow$   choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                      $pos := \lambda$
                                    ifnone
                                      if $\forall x \in \{l, r\}$ SetElement$(x) \vee x = \text{undef}_e$ then
                                        if $l = \text{undef}_e \vee r = \text{undef}_e$ then
                                          $[\![pos]\!] := (undef, undef, \text{undef}_e)$
                                        else
                                          let $v = \{x \mid x \in enumerate(l) \wedge x \notin enumerate(r)\}$ in
                                            $[\![pos]\!] := (undef, undef, setElement(v))$
                                    where
                                      $l \equiv value(\alpha), r \equiv value(\beta)$
```

The Set Union Operator

Set Plugin : union

```
$\llparenthesis {}^{\alpha}\boxed{?} \cup {}^{\beta}\boxed{?} \rrparenthesis_{[650]}$   $\rightarrow$   choose $\lambda \in \{\alpha, \beta\}$ with $\neg evaluated(\lambda)$
                                      $pos := \lambda$
                                    ifnone
                                      if $\forall x \in \{l, r\}$ SetElement$(x) \vee x = \text{undef}_e$ then
                                        if $l = \text{undef}_e \vee r = \text{undef}_e$ then
                                          $[\![pos]\!] := (undef, undef, \text{undef}_e)$
                                        else
                                          let $v = \{x \mid x \in enumerate(l) \vee x \in enumerate(r)\}$ in
                                            $[\![pos]\!] := (undef, undef, setElement(v))$
                                    where
                                      $l \equiv value(\alpha), r \equiv value(\beta)$
```

A.5.5 Math Plugin

Most of the functions provided by the Math plugin are equivalent of their Java counterparts defined in the Java library package `java.lang.Math`. For such functions, we use the descriptions provided by the *Java 2 Platform Standard Edition 5.0 API Specification* [72].

Constants

- `MathE` returns the Number element that is closer in value than any other to e, the base of the natural logarithms.
- `MathPI` returns the Number element that is closer than any other to π, the ratio of the circumference of a circle to its diameter.

Basic Functions

- `abs(v)` returns the absolute value of v.
- `acos(v)` returns the arc cosine of an angle, in the range of 0 through π.
- `asin(v)` returns the arc sine of an angle, in the range of $-\pi/2$ through $\pi/2$.
- `atan(v)` returns the arc tangent of an angle, in the range of $-\pi/2$ through $\pi/2$.
- `atan2(x, y)` converts rectangular coordinates (x, y) to polar (r, θ) and returns θ.
- `cuberoot(v)` returns the cube root of v.
- `cbrt(v)` returns the cube root of v.
- `ceil(v)` returns the smallest (closest to negative infinity) value that is greater than or equal to the argument and is equal to a mathematical integer.
- `cos(v)` returns the trigonometric cosine of an angle.
- `cosh(v)` returns the hyperbolic cosine of v.
- `exp(v)` returns Euler's number e raised to the power of v.
- `expm1(v)` returns $e^v - 1$.
- `floor(v)` returns the largest (closest to positive infinity) value that is less than or equal to the argument and is equal to a mathematical integer.
- `hypot(x, y)` returns $\sqrt{x^2 + y^2}$ without intermediate overflow or underflow.

- `IEEEremainder(v1, v2)` Computes the remainder operation on two arguments as prescribed by the IEEE 754 standard.
- `log(v)` returns the natural logarithm (base e) of v.
- `log10(v)` returns the base 10 logarithm of v.
- `log1p(v)` returns the natural logarithm of the sum of the argument and 1; i.e., $ln(v$ + 1).
- `max(v1, v2)` returns the greater of two values.
- `min(v1, v2)` returns the smaller of two values.
- `pow(x, y)` returns the value of the first argument raised to the power of the second argument.
- `random()` returns a random value with a positive sign, greater than or equal to 0.0 and less than 1.0.
- `round(v)` returns the closest mathematical integer to the argument.
- `signum(v)` Returns zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
- `sin(v)` returns the trigonometric sine of an angle.
- `sinh(v)` returns the hyperbolic sine of v.
- `sqrt(v)` returns the correctly rounded positive square root of v; i.e., $\sqrt{v}$.
- `tan(v)` returns the trigonometric tangent of an angle.
- `tanh(v)` returns the hyperbolic tangent of v.
- `toDegrees(v)` converts an angle measured in radians to an approximately equivalent angle measured in degrees.
- `toRadians(v)` converts an angle measured in degrees to an approximately equivalent angle measured in radians.

Special Functions

- `powerset(set)` computes the powerset of the given set.
- `max({v1,...,vn})` returns the maximum value in a collection of numbers. If there is one non-number in the collection, it returns *undef*.
- `min({v1,...,vn})` returns the minimum value in a collection of numbers. If there is one non-number in the collection, it returns *undef*.

- `sum({v1,...,vn})` returns the sum of a collection of numbers. If there is one non-number in the collection, it returns *undef*.
- `sum({v1,...,vn}, @f)` returns the sum of a collection of numbers, after applying function `f` to the values in the collection. If there is one non-number in the collection, it returns *undef*.
- `powerset({e1,...,en})` returns the powerset of the given set of elements.

Appendix B

CoreASM Examples

B.1 The Railroad Crossing Example

```
CoreASM RailRoadCrossing

use StandardPlugins
use TimePlugin
use MathPlugin

enum Track = {track1, track2}
enum TrackStatus = {empty, coming, crossing}
enum GateSignal = {open, close}
enum GateState = {opened, closed}

function deadline :  Track -> TIME
function trackStatus :  Track -> TrackStatus
function gateSignal :  -> GateSignal
function gateState :  -> GateState

universe Agents = {trackController, gateController, observer, environment}

// Is it safe to open the guard?
  derived safeToOpen = forall t in Track holds
        trackStatus(t) = empty or ( now + dopen) < deadline(t)

derived waitTime = dmin - dclose

init InitRule

rule InitRule = {
    forall t in Track do {
        trackStatus(t) := empty
        deadline(t) := infinity
    }
    gateState:= opened
    dmin:= 5000
    dmax:= 10000
    dopen:= 2000
    dclose:= 2000
    startTime:= now

    program(trackController) := @TrackControl
    program(gateController) := @GateControl
    program(observer) := @ObserverProgram
    program(environment) := @EnvironmentProgram
    program( self ) := undef
}

rule TrackControl = {
    forall t in Track do {
        SetDeadline(t)
        SignalClose(t)
        ClearDeadline(t)
    }
    SignalOpen
}

rule GateControl = {
    if gateSignal = open and gateState = closed then gateState:= opened
    if gateSignal = close and gateState = opened then gateState:= closed
}

rule SetDeadline(x) =
    if trackStatus(x) = coming and deadline(x) = infinity then
        deadline(x) := now + waitTime

rule SignalClose(x) =
    if now >= deadline(x) and now <= deadline(x) + 1000 then
        gateSignal:= close

rule ClearDeadline(x) =
    if trackStatus(x) = empty and deadline(x) < infinity then
        deadline(x) := infinity
```

```
rule SignalOpen =
    if gateSignal = close and safeToOpen then
        gateSignal:= open


// The observer
 rule ObserverProgram =
    seqblock
        print "Time:  " + (( now - startTime) / 1000) + " seconds"
        forall t in Track do
            print "Track " + t + " is " + trackStatus(t)
        print "Gate is " + gateState
        print ""
    endseqblock


// The environment
 rule EnvironmentProgram =
    choose t in Track do {
        if trackStatus(t) = empty then
            if random < 0.05 then {
                trackStatus(t) := coming
                passingTime(t) := now + dmin
            }

        if trackStatus(t) = coming then
            if passingTime(t) < now then {
                trackStatus(t) := crossing
                passingTime(t) := now + 4000
            }

        if trackStatus(t) = crossing then
            if passingTime(t) < now then
                trackStatus(t) := empty
    }
```

B.2 The Surveillance Scenario

```
CoreASM Surveillance_Scenario
```

```
use Standard
use Math
use Options

option Signature.NoUndefinedId strict

/* --- Universes --- */
enum Moves = {N, NW, W, WS, S, SE, E, EN}

enum Direction = {forward, away}
universe Agents = {agent1, agent2, environment}

/* --- Function Definitions --- */
// state of the environment


/* --- Function Definitions --- */
// state of the environment
function posX: Agents -> NUMBER
function posY: Agents -> NUMBER
function bearingError:  Agents -> NUMBER
function rangeError:  Agents -> NUMBER

function observationHistory:  Agents -> LIST
function move:Agents -> NUMBER
function dir:  Agents -> Direction

function bearingErrorRange:  Agents -> NUMBER
function rangeErrorRange:  Agents -> NUMBER

// --- Initial Rule ---
init InitRule

rule InitRule = {
    program(agent1) := @Agent1Program
    program(agent2) := @Agent2Program
    program(environment) := @EnvironmentProgram
    program( self ) := undef

    // initial positions of agents
    posX(agent1) := 0
    posY(agent1) := 0
    posX(agent2) := 15
    posY(agent2) := 10

    dir(agent2) := forward

    // setting error ranges
    bearingErrorRange(agent1) := 3.14 / 20
    rangeErrorRange(agent1) := 2
    bearingErrorRange(agent2) := 3.14 / 20
    rangeErrorRange(agent2) := 4

    // initial values of agent functions
    forall a in {agent1, agent2} do {
        observationHistory(a) := []
        bearingError(a) := 0
        rangeError(a) := 0
    }
}

// --- Agent Programs ---
 rule Agent1Program = {
    RecordObservation(agent2)
    if isInAOI(agent2) then
        SendMessage( "Agent 2 is in the area of interest." )
    if size(observationHistory( self )) > 1 then
        if approaching( self ) then
            print "Agent 1:  Agent 2 is approaching."
}

rule Agent2Program =  {
    RecordObservation(agent1)
    if dir( self ) = forward then
        MoveToward(agent1)
    else
        MoveAwayFrom(agent1)

    if tooClose(agent1) then
        dir( self ) := away
}

rule EnvironmentProgram =
    forall a in {agent1, agent2} do {
        bearingError(a) := bearingErrorRange(a) * (2 * random - 1)
        rangeError(a) := rangeErrorRange(a) * (2 * random - 1)
    }

// --- Auxiliary Rules ---
 rule RecordObservation(a) =
    add [obsRange(self, a), obsBearing(self, a)] to observationHistory(self)

rule SendMessage(msg) =
    "SendMessage(" + msg + ")"

rule Move(dir) = {
    print "agent1:(" + posX(agent1) + ", " + posY(agent1)
            + ") - agent2:(" + posX(agent2) + ", " + posY(agent2) +
")"
    if dir = N then
        posY( self ) := posY( self ) + 1
    else if dir = S then
        posY( self ) := posY( self ) - 1
    else if dir = W then
        posX( self ) := posX( self ) - 1
    else if dir = E then
        posX( self ) := posX( self ) + 1
    else if dir = EN then {
        Move(N)
        Move(E)
    }
    else if dir = NW then {
        Move(N)
        Move(W)
    }
    else if dir = SE then {
        Move(S)
        Move(E)
    }
    else if dir = WS then {
        Move(S)
        Move(W)
    }
}

/* Move towards agent 'a' */
 rule MoveToward(a) =
    let dir = getDirection(
                atan2(posX(agent1) - posX(self), posY(agent1) - posY(self))
                + (2 * random * bearingError(self) - bearingError(self))
              ) in
        Move(dir)

/* Move away from agent 'a' */
 rule MoveAwayFrom(a) =
    let nb = atan2(posX(agent1) - posX(self), posY(agent1) - posY(self))
                + (2 * random * bearingError(self) - bearingError(self))
                - signum(atan2(posX(agent1) - posX(self),
                        posY(agent1) - posY(self))
                    + (2 * random * bearingError(self) - bearingError(self)))
                * MathPI in
        Move(getDirection(nb))

// Compute a move direction based on the given bearing
 rule getDirection(b) =
    return move in
        let bp = abs(b) in {
            if bp < ( MathPI / 8) then
                move:= N
            if abs(bp - MathPI / 4) < ( MathPI / 8) then
                if (b < 0) then
                    move:= EN
                else
                    move:= NW
            if abs(bp - MathPI / 2) < ( MathPI / 8) then
                if (b < 0) then
                    move:= E
                else
                    move:= W
            if abs(bp - (3 * MathPI / 4)) < ( MathPI / 8) then
                if (b < 0) then
                    move:= WS
                else
                    move:= SE
            if abs(bp - MathPI) < ( MathPI / 8) then
                move:= S
        }

/* ----- Derived Functions ----- */

derived bearing(a) = atan2(posX(a) - posX( self ), posY(a) - posY( self
))
```

```
derived range(a) =
    sqrt( pow(posX(a) - posX( self ), 2) + pow(posY(a) - posY( self
), 2))

derived obsBearing(observer, observed) =
    bearing(observed) + bearingError(observer)

derived obsRange(observer, observed) =
    range(observed) + rangeError(observer)

derived isInAOI(a) =
    obsRange( self , a) > 5 and obsRange( self , a) < 12
    and obsBearing( self , a) < ( MathPI / 3)
    and obsBearing( self , a) > ( MathPI / 6)

derived tooClose(observed) =
    obsRange( self , observed) < 12

derived approaching(observer) =
    head( last(observationHistory(observer)))
    < head( nth(observationHistory(observer),
                size(observationHistory(observer)) - 1))
```

Appendix C

Change List

Since August 2009

- semantics of the operators offered by the following plugins is revised such that in binary operators if both operands are *undef* or one is *undef* and the other is a relevant value (depending on the plugin), the evaluation results in *undef*. In unary operators if the operand is *undef* the result of the operation will be *undef*. Of course, if other plugins evaluate the operation to a non-*undef* value, the *undef* value is ignored and the non-*undef* value will be considered as the value of the operation.

  – Bag, List, Number, Predicate, Set, String

  To Do *Specification of operation evaluation in Kernel.*

Bibliography

[1] M. Altenhofen, A. Friesen, and J. Lemcke. Asms in service oriented architectures. *Journal of Universal Computer Science*, 14(12):2034{2058, 2008.

[2] M. Anlauff. XASM { An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 69{90. Springer-Verlag, 2000.

[3] M. Anlauff and P. Kutter. *eXtensible Abstract State Machines*. XASM open source project: http://www.xasm.org.

[4] Jörg Beckers, Daniel Klünder, Stefan Kowalewski, and Bastian Schlich. Direct support for model checking abstract state machines by utilizing simulation. In *ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z*, pages 112{124, Berlin, Heidelberg, 2008. Springer-Verlag.

[5] B. Beckert and J. Posegga. leanEA: A Lean Evolving Algebra Compiler. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 64{85. Springer, 1996.

[6] 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.

[7] Daniel M. Berry. Formal Methods: the very idea | Some thoughts about why they work when they work. *Science of Computer Programming*, 42(1):11{27, 2002.

[8] 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.

[9] Andreas Blass and Yuri Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Transactions on Computation Logic*, 4(4):578{651, 2003.

[10] E. Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *LNCS*, pages 36{64. Springer, 1990.

[11] E. Börger. A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1{14. Springer, 1990.

[12] E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145{160. Springer-Verlag, 2003.

[13] E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237{257, 2003.

[14] 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.

[15] E. Börger, U. Glässer, and W. Müller. The Semantics of Behavioral VHDL'93 Descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500{505, Los Alamitos, California, 1994. IEEE CS Press.

[16] 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.

[17] 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.

[18] 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.

[19] E. Börger and W. Schulte. A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study. *IEEE Transactions on Software Engineering*, 26(10):872{887, October 2000.

[20] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[21] 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.

[22] G. Del Castillo, I. Durdanovic, and U. Glässer. An Evolving Algebra Abstract Machine. In H. Kleine Büning, editor, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95), volume 1092 of *LNCS*, pages 191{214. Springer, 1996.

[23] Matteo Demuru. Modeling cell methabolic mechanisms through Abstract State Machines. Master's thesis, University of Pisa, Italy, February 2008.

[24] D. Diesen. *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages*. Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.

[25] R. Eschbach, U. Gässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, editor, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 242{265. Springer-Verlag, 2000.

[26] R. Eschbach, U. Glässer, R. Gotzhein, 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.

[27] 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.

[28] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. *Fundamenta Informaticae*, pages 71{103, 2007.

[29] R. Farahbod and U. Glässer. Semantic Blueprints of Discrete Dynamic Systems: Challenges and Needs in Computational Modeling of Complex Behavior. In *New Trends in Parallel and Distributed Computing, Proc. 6th Intl. Heinz Nixdorf Symposium, Jan. 2006*, pages 81{95. Heinz Nixdorf Institute, 2006.

[30] R. Farahbod, U. Glässer, E. Bosse, and A. Guitouni. Integrating Abstract State Machines and Interpreted Systems for Situation Analysis Decision Support Design. In *Proc. of the 11th Intl Conf. on Information Fusion (Fusion 2008)*, July 2008.

[31] R. Farahbod, U. Glässer, P. Jackson, and M. Vajihollahi. High Level Analysis, Design and Validation of Distributed Mobile Systems with CoreASM. In *Proceedings of 3rd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008)*. Springer, October 2008.

[32] R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and Validation of the Business Process Execution Language for Web Services. In Wolf Zimmermann and Bernhard Thalheim, editors, *Abstract State Machines 2004. Advances In Theory And Practice: 11th International Workshop (ASM 2004)*, Germany, March 2004. Springer-Verlag.

[33] R. Farahbod, U. Glässer, and M. Vajihollahi. A Formal Semantics for the Business Process Execution Language for Web Services. In Savitri Bevinakoppa et al., editors, *Web Services and Model-Driven Enterprise Information Systems*, pages 144 | 155, Portugal, May 2005. INSTICC Press.

[34] R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract Operational Semantics of the Business Process Execution Language for Web Services. Technical Report SFU-CMPT-TR-2005-04, Simon Fraser University, Feb. 2005. Revised version of SFU-CMPT-TR-2004-03, April 2004.

[35] R. Farahbod, U. Glässer, and M. Vajihollahi. An Abstract Machine Architecture for Web Service Based Business Process Management. *International Journal of Business Process Integration and Management*, 1:279{291, 2007.

[36] R. Farahbod, U. Glässer, and H. Wehn. Dynamic Resource Management for Adaptive Distributed Information Fusion in Large Volume Surveillance. In *Proc. of SPIE Defense & Security Symposium*, March 2008.

[37] R. Farahbod, Uwe Glässer, and G. Ma. Model Checking CoreASM Specifications. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop (ASM'07)*, 2007.

[38] Roozbeh Farahbod. *CoreASM: An Extensible Modeling Framework & Tool Environment for High-level Design and Analysis of Distributed Systems*. PhD thesis, Simon Fraser University, Burnaby, Canada, May 2009. http://roozbeh.ca/downloads/RoozbehFarahbod-PhDThesis.pdf.

[39] Formal Methods laboratory of University of Milan. *Asmeta*, 2006. Last visited June 2008, http://asmeta.sourceforge.net/.

[40] Free Software Foundation. *GNU Lesser General Public License*, 2007. Available electronically at http://www.gnu.org/copyleft/lgpl.html (Last visited in March 2009).

[41] The Apache Software Foundation. *Apache License*, 2004. Available electronically at http://www.apache.org/licenses (Last visited in March 2009).

[42] Martin Fowler. The New Methodology. April 2003. http://martinfowler.com/articles/newMethodology.html.

[43] V. Gervasi and R. Farahbod. JASMine: Accessing java code from CoreASM. In *Proceedings of the Dagstuhl Seminar on Rigorous Methods for Software Construction and Analysis (LNCS Festschrift)*. Springer, 2009 (to be published).

[44] U. Glässer, R. Gotzhein, and A. Prinz. The Formal Semantics of SDL-2000: Status and Perspectives. *Computer Networks*, 42(3):343{358, 2003.

[45] U. Glässer and Q.-P. Gu. Formal Description and Analysis of a Distributed Location Service for Mobile Ad Hoc Networks. *Theoretical Comp. Sci.*, 336:285{309, May 2005.

[46] U. Glässer, Y. Gurevich, and M. Veanes. Abstract Communication Model for Distributed Systems. *IEEE Trans. on Soft. Eng.*, 30(7):458{472, July 2004.

[47] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Prentice Hall, third edition, 2005.

[48] Y. Gurevich. Evolving Algebras. A Tutorial Introduction. *Bulletin of EATCS*, 43:264{284, 1991.

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

[50] Y. Gurevich and J. Huggins. Evolving Algebras and Partial Evaluation. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 587{592, Elsevier, Amsterdam, the Netherlands, 1994.

[51] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917{951, 2001.

[52] Y. Gurevich and N. Tillmann. Partial Updates. *Journal of Theoretical Computer Science*, 336(2-3):311{342, 2005.

[53] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279{295, 1997.

[54] J. Huggins. An offline partial evaluator for evolving algebras. Technical Report CSE-TR-229-95, University of Michigan, 1995.

[55] ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition*. International Telecommunication Union, 2001.

[56] Olav Jensen, Raymond Koteng, Kjetil Monge, and Andreas Prinz. Abstraction using ASM Tools. In A. Prinz, editor, *Proceedings of the 14th International ASM Workshop (ASM'07)*, 2007.

[57] C. W. Johnson. Literate specifications. *Software Engineering Journal*, 11(4):225{237, July 1996.

[58] A. M. Kappel. Executable Specifications Based on Dynamic Algebras. In A. Voronkov, editor, *Logic Programming and Automated Reasoning*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 229{240. Springer, 1993.

[59] Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97{111, 1984.

[60] William Leiserson. *Elegant, efficient LL (k) parser generation*. PhD thesis, Rochester Institute of Technology, Rochester, USA, 2006.

[61] Jens Lemcke and Andreas Friesen. Composing web-service-like abstract state machines (asms). *Services, IEEE Congress on*, pages 262{269, 2007.

[62] George Z. Ma. Model Checking Support for CoreASM: Model Checking Distributed Abstract State Machines Using Spin. Master's thesis, Simon Fraser University, Canada, May 2007.

[63] Daniele Mazzei, Federico Vozzi, Antonio Cisternino, Giovanni Vozzi, and Arti Ahluwalia. A high-throughput bioreactor system for simulating physiological environment. *IEEE Transactions on Industrial Electronics*, 55(9):3273{3280, 2008.

[64] Mashaal A. Memon. Specification language design concepts: Aggregation and extensibility in coreasm. Master's thesis, Simon Fraser University, Burnaby, Canada, April 2006.

[65] Microsoft Corp. *Microsoft .NET Framework*. Last visited Dec. 2006, http://www.microsoft.com/net.

[66] Microsoft FSE Group. *The Abstract State Machine Language*, 2003. Last visited June 2008, http://research.microsoft.com/fse/asml/.

[67] W. Müller, J. Ruf, and W. Rosenstiel. An ASM Based SystemC Simulation Semantics. In W. Müller et al., editors, *SystemC - Methodologies and Applications*. Kluwer Academic Publishers, June 2003.

[68] Regents of the University of California. *BSD Licenses*, 1990-2009. Available electronically at http://en.wikipedia.org/wiki/BSD_licenses (Last visited in March 2009).

[69] Joachim Schmid. *AsmGofer*, 2008. Available electronically at http://www.tydo.de/Doktorarbeit/AsmGofer/ (Last visited in July 2008).

[70] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18{21, 1975.

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

[72] Sun Microsystems, Inc. *The Java 2 Platform Standard Edition 5.0 API Specification*. Sun Microsystems, Inc., 2004. (http://java.sun.com/j2se/1.5.0/docs/api).

[73] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949 of *Lecture Notes in Computer Science*, pages 39{76. Springer, 2008.