Design and Specification of the CoreASM Execution Engine

Part 1: The Kernel

R. Farahbod[1], V. Gervasi[2], U. Glässer[1], and M. Memon[1]

[1] Computing Science, Simon Fraser University, Burnaby, B.C., Canada
{rfarahbo,glaesser,mamemon}@cs.sfu.ca
[2] Dipartimento di Informatica, Università di Pisa, Italy
gervasi@di.unipi.it

Technical Report SFU-CMPT-TR-2006-09
May 2006

License and Copyright Notice

Abstract

The CoreASM project, an attempt to make *abstract state machines* (ASMs) executable, was first introduced in [16, 15]. The aim of this project is to specify and implement an extensible ASM execution engine for an extensible language that is as close as possible to the mathematical definition of pure ASMs. This document focuses on the design of the Kernel of the CoreASM engine and the bare essential structures of the CoreASM language and provides an update on the latest achievements and improvements.

Acknowledgement

Our sincere appreciation to **Egon Börger** for many inspiring discussions and persistent encouragement of the CoreASM project, as well as his valuable feedback on an early draft version of this paper. We would also like to thank **George Ma** for his comments on and contribution to the implementation of the CoreASM Kernel.

Contents

Chapter 1

Introduction

Abstract state machines are well known for their versatility in modeling complex architectures, languages, protocols and virtually all kinds of sequential and distributed systems with an orientation toward practical applications. The particular strength of this approach is the flexibility and universality it provides as an abstract mathematical framework for semantic modeling of functional requirements. This is invaluable when used to bridge the gap between informal requirements and precise specifications, for instance, in the earlier phases of system design and during reverse engineering of requirements from implementations. This usage of ASMs has extensively been studied by researchers and developers in academia and industry, leading to the establishment of a solid methodological foundation providing practical guidelines for building ASM ground models. Widely recognized applications include semantic foundations of industrial system design languages like the ITU-T standard for SDL [22], the IEEE language VHDL [6], programming languages like JAVA [28] and C# [5], communication architectures, etc.

The research project we describe here focuses on the design of a lean, executable ASM language, called CoreASM, in combination with a supporting tool environment for high-level design, experimental validation and formal verification (where appropriate) of abstract system models. We concentrate on control-intensive software systems, especially, distributed and embedded systems and related system design languages; we also consider sequential languages and synchronous systems, and, to some extent, hardware related aspects. Specifically, we are developing a platform-independent *engine* for executing the CoreASM language and a graphical user interface (GUI) for interactive visualization and control of CoreASM simulation runs. The engine comes with a sophisticated and well defined interface and thereby enables future development and integration of complementary tools (e.g., for symbolic model checking and automated test case generation).

Exploring the problem space for the purpose of writing an initial specification calls for a language that emphasizes freedom of experimentation and supports easy modifiability. Moreover, such a language must support writing highly abstract and concise specifications by minimizing the need for encoding in mapping the problem space to a formal model. In our work we address the needs of that part of the software development process that is closest to the problem space, as illustrated in Figure 1.1.

Model-based systems engineering naturally demands for abstract executable

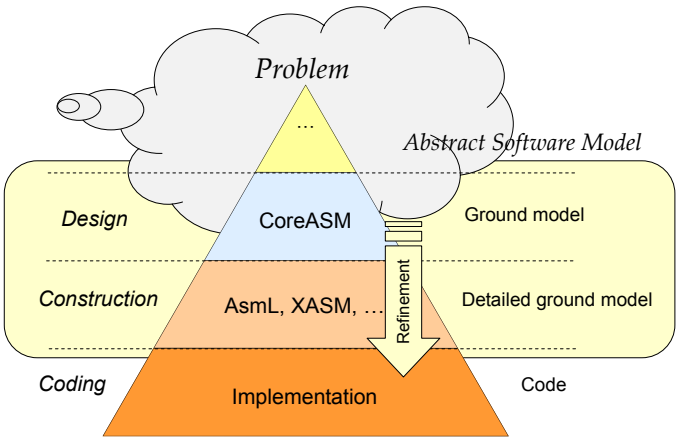

Figure 1.1: Background and Motivation

specifications as a basis for experimental validation through simulation and testing. Thus it is not surprising that there is a considerable variety of executable ASM languages (see [9], Section 8.3) that have been developed over the years. The most prominent one is AsmL (ASM Language)[25], developed by the FSE group at Microsoft Research. AsmL is an executable language based on the concept of ASMs but also incorporates numerous object oriented features, thus departing in this respect from the theoretical model of ASMs, and comes with the richness of a fully fletched programming language. It also lacks any built-in support for dealing with distributed systems. Its design was shaped by the practical needs of dealing with fairly complex requirements and design specifications for the purpose of software testing; it can be thus said that its primary concerns are toward the world of code. This has made it less suitable for initial modeling at the peak of the problem space and also reduces the freedom of experimentation.

The CoreASM language and tool architecture focus on early phases of the software design process, and CoreASM primary concerns are toward the world of problems. In particular, we want 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 specification technique that has been exploited in [9]. In this process, we aim at maintaining executability of even fairly abstract models. Another important characteristic that differentiate 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 [18] 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 proved so fruitful in the development of ASM concepts, and to this end we designed our architecture around the concept of *plug-in*s that allows to customize the language to specific needs.

An extensible, platform independent tool package (the language, its engine,

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

and the GUI) will be an asset both for industrial engineering of complex software systems by making software specifications and designs more robust and reliable, and for research by providing facilities for the testing of proposed extensions to the basic ASM language.

Chapter 2

Architecture Overview

The CoreASM engine consists of four components: a *parser*, an *interpreter*, a *scheduler*, and an *abstract storage* (Figure 2.1). 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.

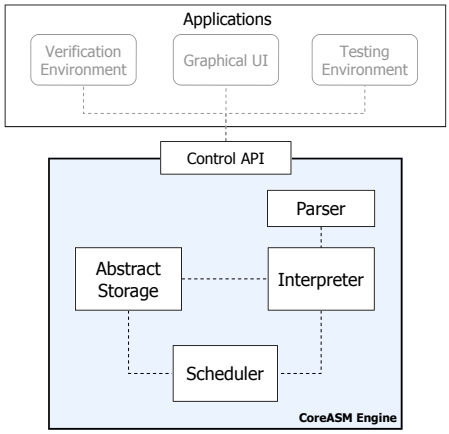

Figure 2.1: Overall Architecture of CoreASM

The parser reads a CoreASM specification and provides the interpreter with an annotated parse tree for each program. The interpreter then evaluates the programs in the specification by examining all the rules and generating update sets. The abstract storage manages the data model for the abstract state. In particular, it stores the current state of the simulated machine along with the history of its previous states, which can be used to examine the run traces or to rollback to a previous state and resume the computation. The number of possible rollbacks is configurable.[1] 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. The role of the scheduler is to orchestrate the whole

[1]It is important to mention that the rollback mechanism does not rollback the environment.

execution process. In particular, for distributed ASMs the scheduler is responsible for selecting the set of agents that will contribute to the next computation step and coordinating the execution of those agents. The scheduler also manages cases of inconsistency of update sets generated in a step.

The execution process of a single step in the CoreASM engine is as follows (refer also to Figures 2.6 to 2.9 in Section 2.2):

1. The Control API sends a STEP command to the scheduler.
2. The scheduler gets the whole set of agents from the abstract storage (from the special set *agents*).
3. The scheduler selects a subset of these agents, which will perform computation in the next step.
4. The scheduler selects a single agent from this set and assigns it to the special variable *self* in the abstract storage.
5. The scheduler then calls the interpreter to run the program of the current agent (retrieved by accessing *program(self)* in the current state).
6. The interpreter evaluates the program.[2]
7. When evaluation is complete, the interpreter notifies the scheduler that the interpretation is finished.
8. The scheduler then selects another agent in the selected set of agents. If there are no more agents left in the set, the scheduler calls the abstract storage to fire the accumulated updates.
9. The abstract storage notifies the scheduler whether the update set has any conflicts or it was successfully fired. This notification can lead to selection of a different subset of agents to be executed in the step, or can be sent back to the Control API.

2.1 CoreASM Components

In this section we present in more detail the basic components of the CoreASM engine, together with their extensibility mechanisms. The architecture is partitioned along two dimensions (see Figure 2.2). The first one, that we already presented, identifies the four main modules (parser, interpreter, scheduler, abstract storage) and their relationships. The second dimension, that we will discuss in Section 2.3, distinguishes between what is in the *kernel* of the system — thus implicitly defining the extreme bare bones ASM model — and what is instead provided by extension plug-ins.

The reader may notice that these two dimensions correspond to what in the ASM literature have been called *modular decomposition* and *conservative refinement* respectively. In particular, our plug-ins progressively extend in a conservative way the capabilities of the language accepted by the CoreASM engine, in the same

[2]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.

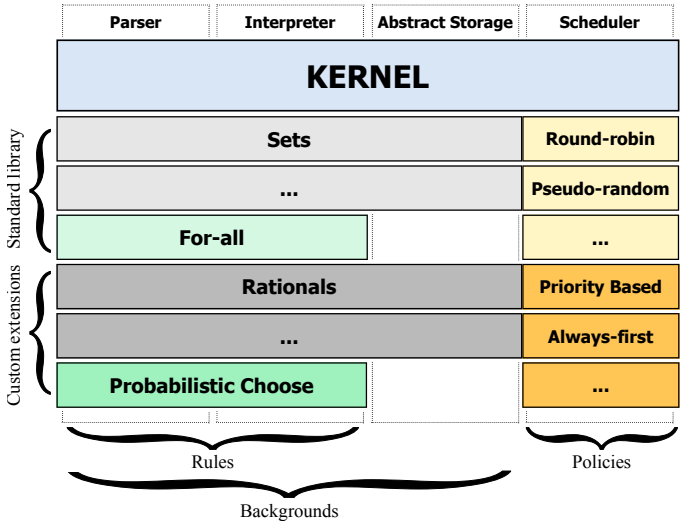

Figure 2.2: Layers and Modules of the CoreASM Engine

spirit in which successive layers of the Java [28] and C# [5] languages have been used to structure the language definition into manageable parts.

The first module in our architecture is the parser. The parser generates annotated abstract syntax trees for rules and programs of a given CoreASM specification. Each node in these trees may have a reference to the plug-in where the corresponding syntax is defined. For example in Figure 2.3, there are nodes that belong to the backgrounds of sets and 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 second module, the interpreter, executes programs and rules, possibly calling upon background plug-ins to perform expression evaluation, and upon rules plug-ins to interpret certain rules. 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 plug-ins to generate the actual updates (as will be described in more detail on page 17)[3]. 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 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 from a universe Element. The abstract storage also provides interfaces to retrieve values from a given location in the current state and to apply updates.

In addition, it also provides other auxiliary information about the locations of current state, such as the ranges and domains of functions or the background to which a particular function or value belongs to.

Finally, the scheduler orchestrates every computation step of an ASM run. In a

[3]Where no confusion can arise, in the following we use the generic term "updates" to refer both to actual updates and to update instructions.

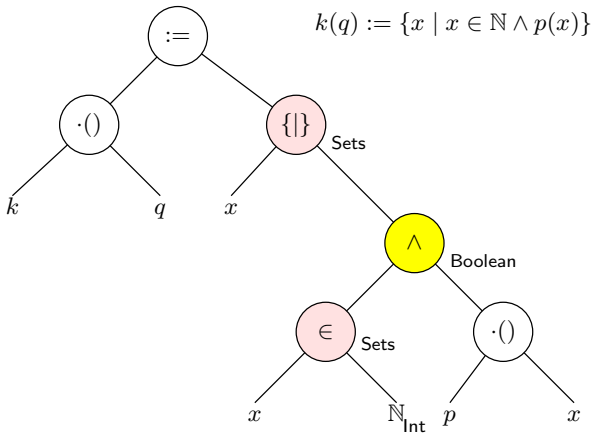

Figure 2.3: Sample Annotated Parse Tree

sequential 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 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 [9], 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 computation of the next update set. The scheduler directly interacts with the abstract storage to retrieve the current set of DASM 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.

2.2 Engine Life-cycle

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

1. Initializing the engine (Figure 2.4)
   (a) Initializing the kernel
   (b) Loading the plug-ins library catalogue
   (c) Loading and activating plug-ins from a standard library
2. Loading a CoreASM specification (Figure 2.5)
   (a) Parsing the specification header
   (b) Loading further needed plug-ins as declared in the header
   (c) Parsing the specification body
   (d) Initializing the abstract storage
   (e) Setting up the initial state

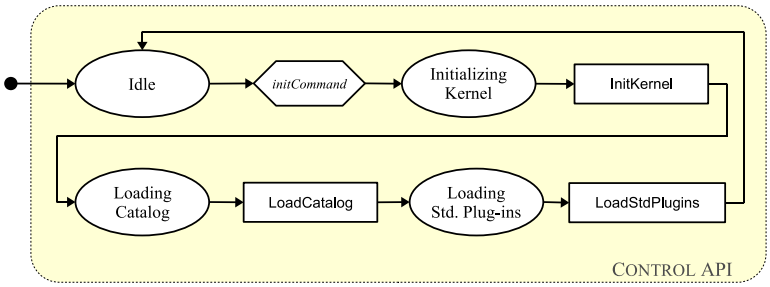

Figure 2.4: Control State ASM of Initializing CoreASM Engine

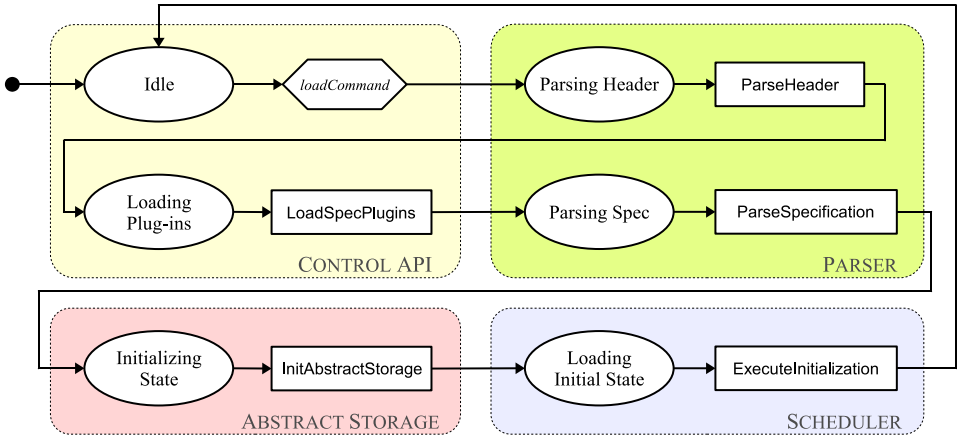

Figure 2.5: Control State ASM of Loading a CoreASM Specification

3. Execution of the specification

   (a) Execute a single step

   (b) If termination condition not met, repeat from 3a

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). See Appendix A for the definition of macros in Figures 2.4 and 2.5.

In the following we present a high-level but precise specification of the execution process (step 3a above) which was presented informally at the beginning of this section. The structure of the specification is that of a control state ASM, as shown in Figures 2.6 to 2.9. 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.

The engine starts its execution in the *Idle* state of the Control API module (Figure 2.6). In this state, the engine simply waits for a *STEP* command from

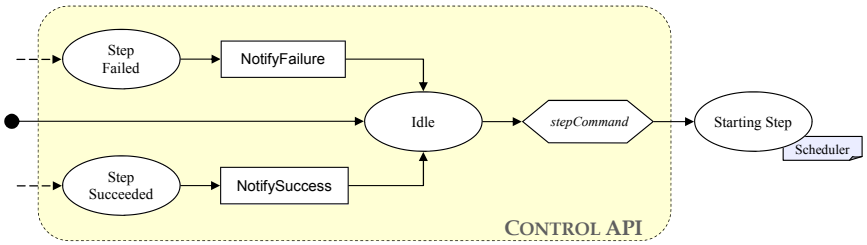

Figure 2.6: Control State ASM of a STEP command: Control API Module

the environment[4] (e.g., an interactive GUI or a debugger), to start the actual computation; this is performed by changing the state to *Starting Step* which then transfers the control flow to the scheduler.

The StartStep rule in the scheduler simply initializes *updateInstructions* (the multiset of accumulated update instructions for the step), *agentSet* (the current set of agents of the simulated machine), and *selectedAgentsSet* (the set of agents selected to perform computation in the current step). The latter is then assigned a value in the RetrieveAgents rule by querying the abstract storage module for the current value of *agents* in the simulated machine. 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. The state is then changed to *Selecting Agents*.

Scheduler

StartStep ≡
  updateInstructions := {|}
  agentSet := *undef*
  selectedAgentsSet := {}

RetrieveAgents ≡
  agentSet := $getValue((\text{“agents”}, \langle\rangle))$

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. 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 is moved back to *Choosing Agent* until all selected agents have been executed.

[4]The Control API provides several other commands that are needed to implement a complete execution environment; we restrict ourselves to the most basic *STEP* command here to keep the presentation manageable.

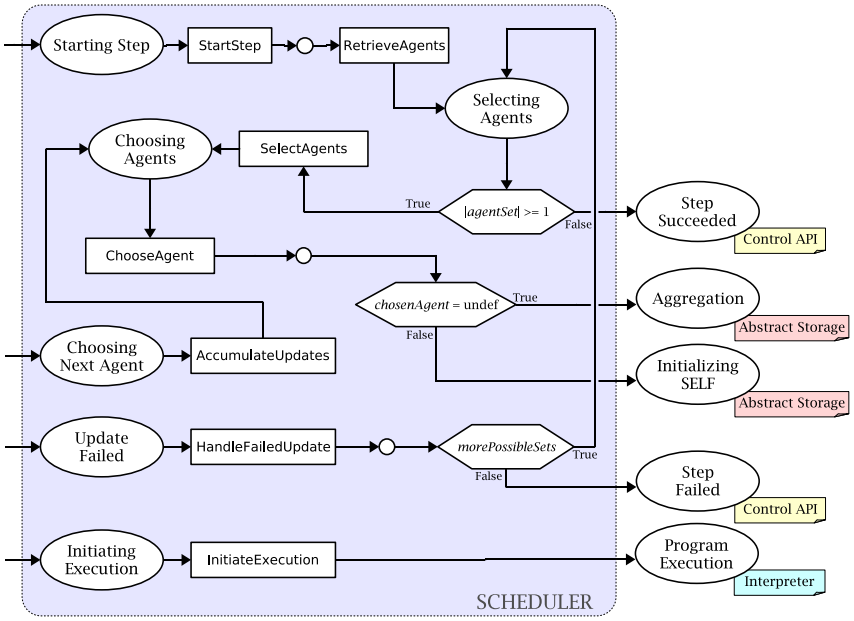

Figure 2.7: Control State ASM of a STEP command : Scheduler

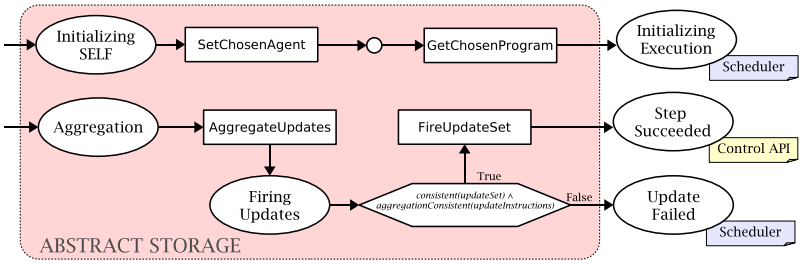

Figure 2.8: Control State ASM of a STEP command : Abstract Storage

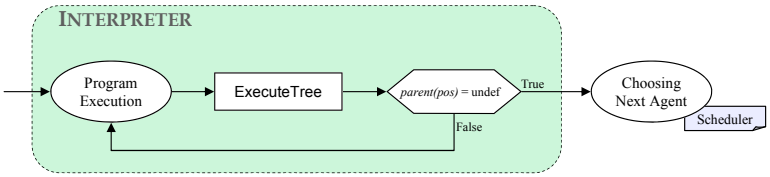

Figure 2.9: Control State ASM of a STEP command : Interpreter

Scheduler

SelectAgents ≡
  choose s **with** $s \subseteq agentSet \wedge |s| \geq 1$ **do**
    $selectedAgentsSet := s$

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 (by assigning it to the special variable *self* in the simulated state) 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 the *Initiating Execution* state.

Abstract Storage

SetChosenAgent ≡
  SetValue((“self”, ⟨⟩), *chosenAgent*)

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

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 [28], we interpret a program by associating values, updates and locations to nodes in the abstract syntax 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 ≡
  pos := root(chosenProgram)
  InitProgramExecution   // in the interpreter
```

The specification of the interpreter is explored in more detail in Section 3.5. 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 plug-ins to delegate interpretation of the associated extensions. As already discussed earlier, nodes of the parse tree corresponding to grammar rules provided by a plug-in are annotated with the plug-in identifier; here we abstract from the details of how this annotation is implemented, and use instead an oracle function *plugin*(*node*) for this purpose. If a node is found to refer to a plug-in, rules provided by that plug-in are obtained through the *pluginRule* function and executed; otherwise, the kernel interpreter rules (see Section 3.5) are used. Results of the interpretation of node *pos* are stored alongside the node, and accessed by three functions, namely *value*(*pos*) will return the computed value for an expression node, *updates*(*pos*) will return the set of updates generated by a rule node, and *loc*(*pos*) will return the location denoted by the node (which is used as lhs-value for assignments). Section 3.5.2 presents a more precise definition of these functions.

Interpreter

```
InitProgramExecution ≡
  ClearTree(root(chosenProgram))
  ClearEnvironmentVariable

ExecuteTree ≡
  if ¬evaluated(pos) then
    if plugin(pos) ≠ undef then
      let R = pluginRule(plugin(pos)) in
        R
    else
      KernelInterpreter
  else
    if parent(pos) ≠ undef then
      pos := parent(pos)
```

After executing the programs of all the agents selected in the *Selecting Agents* state, 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 (through the AggregateUpdates rule), checks the consistency of the computed updates (possibly interacting with the relevant background plug-ins to evaluate equality), and either applies the updates to the current state by FireUpdateSet (thus obtaining the next state), or provides an indication of failure by changing the state to *Update Failed*.

Abstract Storage

AggregateUpdates ≡
  let $ap = \{a \mid a \in \text{PLUGIN} \wedge aggregator(a)\}$ **in**
    $updateSet := \bigcup_{p \in ap}$ InvokeAggregation$(p,\ updateInstructions)$

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

If an inconsistent set of updates is generated in a step, the HandleFailedUpdate rule in the scheduler module selects a different subset of agents for execution, and the step is re-initiated. The process is iterated until a consistent set of updates is generated, in which case the computation proceeds in the *Step Succeeded* state of the Control API, or all possible combinations have been exhausted, in which case the *Step Failed* state is entered instead. 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 the NotifySuccess or the NotifyFailure rule in 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).

2.3 Plug-ins

In keeping with the micro-kernel spirit of the CoreASM approach, most of the functionality of the engine is implemented through plug-ins to a minimal kernel. The architecture supports three classes of plug-ins: *backgrounds*, *rules* and *policies*, whose function is described in the following.

- Background plug-ins provide all that is needed to define and work with new backgrounds, namely (i) an extension to the parser defining the concrete syntax (operators, literals, static functions, etc.) needed for working with elements of the background; (ii) an extension to the abstract storage providing encoding and decoding functions for representing elements of the background for storage purposes, and (iii) an extension to the interpreter providing the semantics for all the operations defined in the background.

- Rule plug-ins are used to implement specific rule forms, with the understanding that the execution of a rule always results in a (possibly empty) set of updates. Thus, they include (i) an extension to the parser defining the concrete syntax of the rule form; (ii) an extension to the interpreter defining the semantics of the rule form.

- Policy plug-ins are used to implement specific scheduling policies for 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[5]. It is worthwhile to note that only a single scheduling policy can be in force at any given time,

[5]The policies in these plug-ins can also be called upon for implementing the **choose**-rule; an extension plug-in provides an enhanced version of **choose** that allows the specifier to explicitly state which policy to use.

whereas an arbitrary number of background and rule plug-ins can be all in use at the same time.

The plug-in framework is further discussed in Section 3.6. Plug-ins are characterized by an abstract interface which is used by the CoreASM engine to communicate with the plugin (see Section 3.6).

In CoreASM, the kernel (see Figure 2.2) 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, boolean algebra ($\wedge$, $\vee$, $\neg$, etc.) is not, and is instead provided through a background plug-in. In the same vein, while universes are represented in the kernel through set characteristic functions, the background of finite sets is implemented in a plug-in, which provides expression syntax for defining them (see the example in Figure 2.3), 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 status due to its privileged access to the Reserve. All other rule forms (e.g., **if**, **choose**, **forall**), as well as sub-machine calls and macros, are implemented as plug-ins in a standard library, which is implicitly loaded with each CoreASM specification.

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.

In addition to modular extensions of the engine, plug-ins can also register themselves for *Extension Points*. Each mode transition in the execution engine is associated to an extension point. At any extension point, if there is any plug-in registered for that point, the rule provided by the plug-in at registration time is executed before the engine proceeds into the new mode. Such a mechanism enables extensions to the engine's life-cycle which facilitates implementing various practically relevant features such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the behavior of the simulated machine (e.g., coverage analysis or profiling). A plug-in, 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 already mentioned, the CoreASM engine is accompanied by a *standard library* of plug-ins including the most common backgrounds and rule forms (i.e., those defined in [9]), an extension library including a small number of specialized backgrounds and rules, and by a set of specifications for writing new plug-ins that

can easily be integrated in the environment. Extension plug-ins must be explicitly imported into an ASM specification by an explicit **use** directive.

2.4 Control API

Code Reference:

- `org.coreasm.engine.CoreASMEngine`
- `org.coreasm.engine.ControlAPI`

The following functions and rules define the interface of the engine to its external environment:

- Engine initialization
  - **rule** Initialize
    initializes the engine as presented in Figure 2.4.
  - **rule** LoadSpecification(*spec*)
    loads a new CoreASM specification into the engine (see Figure 2.5).
  - *spec* : Spec
    returns the current CoreASM spec that the engine is working on.
- State of the engine
  - *state* : State
    returns the current state (after last computation step) of the engine.
  - *prevState* : Integer → State
    returns the last i'th state of the engine. For $i = 0$, this function returns the current state.
  - *updateSet* : Integer → UpdateSet
    returns the last i'th update set of the engine. For $i = 0$ this function returns the last update set.
  - *updateInstructions* : UpdateInst-multiset
    returns the update instructions computed in the last step of the engine.
  - **rule** SetState(*newState* : State)
    assigns *newState* as the current state of the engine. After a successful execution of this rule, *state* returns the *newState*.
  - **rule** UpdateState(*update* : UpdateSet)
    updates the current state by applying the given set of updates.
- *engineProperties* : Name → Name
  holds all the defined engine properties and their values. The behaviour of the engine can be customized by these properties. An example of such properties could be:
  - *engineProperties*("TypeChecking") ∈ { "Ignore" , "Warning" , "Error" }
    default is "Ignore"

- *engineMode* : EngineMode
  returns the current execution mode of the engine.

- **rule** HardInterrupt
  sends a hard interrupt signal to the engine (changes a flag). If the engine is computing a step, it will try to interrupt the current computation. The engine then goes to an interrupted mode. If the engine is not running (not computing a step), this rule does nothing.

- **rule** SoftInterrupt
  sends a soft interrupt signal to the engine (like changing a flag). If the engine is computing a series of steps, it will stop after computation of the current step. The engine stops normally. This rule is meant to stop a series of computations triggered by executing Run. If the engine is not running (not computing a step), this method does nothing.

- **rule** Step
  performs one computation step. As a result of the execution of this rule, the values of *state*, *prevState*, *updateSet*, and *engineMode* may be changed.

- **rule** Run(i : INT)
  performs a specified number of computation steps. For i equal to zero, the engine runs until it is interrupted or an error occurs. As a result of the execution of this rule, the values of *state*, *prevState*, *updateSet*, and *engineMode* may be changed.

- *eventObservers* : EventObserver-set
  is a set of event observers (see the *Observer* design pattern in [17]) that will be notified by the engine in case of the occurrence of the following events[6]:

  – Error messages generated
  – Warning messages generated
  – Standard output generated (after a step)
  – Standard input required
  – Computation events (e.g., engine interrupted or completed a step)

  Applications are supposed to **add** or **remove** observers to this set.

The CoreASM engine also provides internal services to engine components. The following rules define the internal interface of the engine (Control API):

Not fully implemented yet.

- **rule** Error(*msg* : Name)
  reports an internal error to the outside environment. The Error rule used in the ASM specification of the interpreter may use this rule.

Error

```
Error(msg : NAME) ≡
  forall observer in eventObservers(self) with isErrorObserver(observer)
    observer.errorOccured(msg)
```

[6] We may think of more events in the future.

- **rule** Warning(*msg* : Name)
  sends a warning message to the environment. The Warning rule used in the ASM specification of the interpreter may use this rule.

Warning

```
Warning(msg : NAME) ≡
  forall observer in eventObservers(self) with isWarningObserver(observer)
    observer.warningOccured(msg)
```

- **rule** StdOutput(*text* : Name)
  basically informs the environment of the engine of a change to the value of *output*. The idea is to let the environment monitor the output of the simulated machine while the engine is performing a sequence of steps.

StdOutput

```
StdOutput(text : NAME) ≡
  forall observer in eventObservers(self) with isOutputObserver(observer)
    observer.outputGenerated(msg)
```

- **rule** StdInputRequest
  informs the environment of the engine that the simulated machine needs an input from the environment. The input value then can be assigned to the standard input variable (*input*) in the state of the machine.

StdOutput

```
StdInputRequest ≡
  forall observer in eventObservers(self) with isInputObserver(observer)
    observer.inputRequested
```

Chapter 3

The Kernel

This chapter explores the kernel of the CoreASM engine. We first focus on special updates and the aggregation mechanisms. We then look into the four components of the kernel and reveal their underlying architecture. We then revisit special updates and introduce update composition mechanisms which are useful in Turbo ASM constructs. An overview of the plug-in framework of the engine concludes this chapter.

3.1 Aggregation and Special Updates

In this section, we introduce special updates and aggregation as part of the CoreASM engine architecture. In particular we discuss how the embodiment of aggregation affects the engine architecture and a CoreASM step as a whole.

3.1.1 Rules and Their Side Effects

As each rule of a CoreASM specification is executed by the Interpreter, it is expected to produce a (potentially empty) set of updates, each update being viewed as a 2-tuple expected to consist of a location and a value:

$$\langle \textsc{Loc}, \textsc{Element} \rangle$$

The union of all of these sets returned by rules during a single step of the simulated machine constitute the update set for a CoreASM step.

However, the possibility of having elements within the simulated machine which are themselves based on axioms and structure (e.g. sets, maps, trees) and whose internal structures may also be updated by rules, requires the CoreASM have facilities to handle such incremental updates. Recall that Gurevich's partial updates consist of a location and a particle. The particle represents the partial modification to be made to the element at the given location; the particle is a mathematical function in which the entire incremental change is encoded. Notice that the essential function of a particle in a partial update is to represent the type of incremental change to perform, and a value associated to that change.

To accommodate the representation of incremental change into CoreASM, we allow rules to return *update instructions*, rather than updates; like updates, they consist of location and value, but they also include an *action* to be performed on

the element at the the given location. Update instructions are viewed as a 3-tuple of the form:

$$\langle \textsc{Loc}, \textsc{Element}, \textsc{Action} \rangle$$

The combination of the value and action represent the intended incremental modification to be made to the element residing at the given location. Update instructions containing incremental modification actions are referred to as *special updates*. Going back to our example, the special update resulting in the addition of element msg_x to a set at location l would produce an update instruction of the form:

$$\langle l, msg_x, setAddAction \rangle$$

where $setAddAction \in \textsc{Action}$ and Action is the domain of all actions supported by the simulated machine.

Regular Updates

For the sake of homogeneity we require the **update**-rule to return update instructions as well. However the action for such *regular updates* is always $updateAction \in \textsc{Action}$.[1]

$$\langle \textsc{Loc}, \textsc{Element}, updateAction \rangle$$

Update and Update Instruction

When discussing ASMs, the term *update* is typically used to refer to both the act of modifying a location, as well as the data structure representing an update. With the introduction of the *update instruction*, when discussing the CoreASM machine we use the term update to refer to the act of modifying a location, whereas the term update instruction is used to refer to the data structure representing an update of any kind (i.e. regular or special update); however, at times we also use these terms interchangeably when the difference between them is irrelevant.

3.1.2 Update Instruction Notation

All update instructions have the following functions defined over them:

- $\langle\langle \cdot \rangle\rangle : \textsc{UpdateInst} \rightarrow \textsc{Loc} \times \textsc{Element} \times \textsc{Action}$ holds the constituents of the update instruction given by a triple formed by a location, a value, and an action to be performed. We access elements and establish properties of such triples through the following derived functions:
  - $uiLoc : \textsc{UpdateInst} \rightarrow \textsc{Loc}$ returns the location associated with the given update instruction, i.e. $uiLoc(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 1$.
  - $uiVal : \textsc{UpdateInst} \rightarrow \textsc{Element}$ returns the value associated with the given update instruction, i.e. $uiVal(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 2$.
  - $uiAction : \textsc{UpdateInst} \rightarrow \textsc{Action}$ returns the action associated with the given update instruction, i.e. $uiAction(ui) \equiv \langle\langle ui \rangle\rangle \downarrow 3$.

[1] This follows Gurevich's approach, where a total update to a location results in a partial update containing an `overwrite` particle.

- *aggStatus* : UpdateInst × Plugin → Flag indicates the aggregation status of an update instruction, as set by a given aggregator plug-in; Flag = {*successful*, *failed*}. If an update instruction has not been processed by a plug-in, *undef* is returned. Note that the purpose this function will become more clear in subsequent sections.

3.1.3 A CoreASM Step

The computation of a single step of an ASM program can be summarized very simply as follows:

1. Execute the program rule and collect the updates into a set.
2. If update set is consistent, apply the updates.

However, with the introduction of special updates, the process of creating the final update set requires additional work. Update instructions are collected into an *update multiset* stored in the function *updateInstructions*. Once the execution of the program rule is complete, all update instructions pertaining to a particular location are *aggregated* into one single update per location. *Aggregation* is the process of combining all update instructions affecting a single location of a machine into one single update called the *resultant update*. The *aggregation phase* of a CoreASM step performs aggregation on all locations affected by the step. Note that resultant updates cannot and should not depend on the order in which all update instructions for a location are combined, as all updates producing them occur simultaneously according to ASM semantics; we shall explain this further in Section 3.1.6.

It is important to highlight the difference between regular, resultant, and basic updates. A *regular* update is a typical ASM update produced by an **update**-rule, whereas a *resultant* update is an ASM update produced by aggregating all special update instructions and all regular updates into one unified change affecting a single location of the machine. The word "resultant", "regular" will be dropped when its' meaning is obvious from context. An update is *basic* if every update operating on its location is regular, thus implying no aggregation need be performed on its location.

The aggregation phase results in an update set, consisting of basic updates and resultant updates. The traditional ASM step augmented with the aggregation phase is summarized as follows:

1. Execute the program rule and collect the update instructions into a multiset.
2. Aggregate the update instructions in the multiset, producing the update set.
3. If the aggregation phase is successful and update set is consistent, apply the updates.

When control is in the Scheduler (in the *Choosing Agents* mode) and all agents selected to execute in a step have been executed, control now moves to the *Aggregation* mode in Abstract Storage. Here the rule AggregateUpdates (which is formally defined in the next section) performs the aggregation of the multiset *updateInstructions*. When aggregation is complete, control moves to the *Firing Updates* mode where both update set consistency and aggregation consistency are confirmed before application of the update set (see Figures 2.7 and 2.8).

3.1.4 Responsibility for Aggregation

Background plug-ins, which extend CoreASM with a background class should provide all that is necessary to manipulate elements which originate from their background[2]. For backgrounds that consist of elements with internal structure that can be manipulated, background plug-ins provide rule forms that result in special update instructions, as well as provide an algorithm for aggregation. We call these plug-ins *aggregators* or *aggregator plug-ins*.

We say that an aggregator plug-in is responsible for:

- An action other than the *updateAction* action (see Section 3.1.5), if it is equipped to handle its aggregation.
- Aggregation of a given update instruction if the update instruction:
  - Contains an action for which the plug-in is responsible.
  - Contains an *updateAction* (making it a regular update) and there is another update instruction which it is responsible for that also operates on the the same location.
- A location if update instructions operating on that location are its responsibility.

Upon being called for aggregation, a plug-in will aggregate all update instructions for which it is responsible, flagging those update instructions it has processed. It is important to note that the order in which plug-ins are called to perform aggregation does not affect the resultant updates produced. Also note that the failure in aggregation of a single plug-in will not foil the aggregation attempts of other plug-ins. Upon completion of the aggregation phase, an update set is created from the union of resultant updates.

Abstract Storage

```
AggregateUpdates
  updateSet ← Aggregate(updateInstructions)

Aggregate(uMset : UpdateMultiset) ≡
  let ap = {a | a ∈ Plugin ∧ aggregator(a)} in
      forall p ∈ ap do
        resultantUpdates(p, uMset) ← InvokeAggregation(p, uMset)
    seq
      // Results in an update set
      result := ⋃_{p∈ap} resultantUpdates(p, uMset)

InvokeAggregation(p : Plugin, uMset : UpdateMultiset) ≡
  let R = aggregatorRule(p) in
    result ← R(uMset)
```

The *resultantUpdates* function is used to collect resultant updates from plug-ins for a given multiset of update instructions, and the *aggregatorRule* function is expected

[2]While we expect a plug-in providing a background class to provide all that is necessary to manipulate elements of its background, there may be cases where it is more appropriate for functionality to be present in different plug-ins. Thus, we do not enforce this expectation.

to return the rule implementing the aggregation algorithm of the given plug-in. Note that the parameterized rule for aggregation, Aggregate, may be called on any update multiset. Also note that in InvokeAggregation, a plug-in aggregator rule is expected to accept a multiset as an argument, and its invocation should cause the return of its resultant updates.

3.1.5 Basic Update Aggregator

The keen observer will have noticed that once all aggregator plug-ins have completed aggregation successfully, the resultant update set will not contain basic updates (i.e. regular updates for locations which do not require aggregation). The *Basic Update Aggregator* solves this problem by masquerading as an aggregator plug-in and returning a set of all regular updates for locations which do not require any aggregation. It is defined as follows

Abstract Storage

BasicUpdateAggregator($uMset$: UpdateMultiset) ≡
  result := { }
 seq
  forall $ui \in uMset$ **with** $uiAction(ui) = updateAction$ **do**
   if $\nexists\, ui' \in uMset,\ uiLoc(ui) = uiLoc(ui') \wedge uiAction(ui') \neq updateAction$ **then**
    add ui **to result**
    $aggStatus(ui, buPlugin) := successful$

where *buPlugin* represents the Basic Update Aggregator as a plug-in, $buPlugin \in$ Plugin and $aggregator(buPlugin) = true$. Evaluation of $aggregatorRule(buPlugin)$ results in the rule BasicUpdateAggregator. The Basic Update Aggregator is called by Aggregate along with all aggregator plug-ins. Note that the Basic Update Aggregator flags all update instructions it processes with *successful*.

3.1.6 Plug-in Aggregation Consistency

While a plug-in is performing its aggregation on the multiset, it may encounter a situation where the update instructions for a given location that it is responsible for cannot be aggregated into a regular update. Such a situation occurs when one of the following holds:

- There are update instructions which make no semantic sense in context[3]. (e.g. the addition of an element to a set, on a location which contains no set element in the current state).
- The result of aggregation of a location depends on the order in which special update instructions for that location are combined. Recall that since special updates resulting from a single step of the machine occur the same time according to ASM semantics, the result of their aggregation must not be ambiguous for their aggregation to be consistent.

When the aggregation of all update instructions affecting a given location is deemed inconsistent, the following rule is called by the plug-in to flag all updates to the location as *failed*:

[3]Acceptable semantics of special updates, and the aggregation resulting from their update instructions, are defined by the aggregation algorithm which processes them.

Abstract Storage

HandleInconsistentAggregation(l : Loc, $uMset$: UpdateMultiset, p : Plugin) ≡
  forall $ui \in uMset$ **with** $uiLoc(ui) = l$ **do**
    $aggStatus(ui, p) := failed$

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

3.1.7 Aggregation Algorithms Provided

There are very few hard-and-fast requirements on the algorithm provided by an aggregator plug-in. It is expected to:

- Aggregate all update instructions in the update multiset that it is responsible for, and return the set of all its resultant updates.
- Determine if aggregation on a given location will result in inconsistency, and handle such inconsistencies appropriately.
- Flag all update instructions considered during its aggregation as either *successful* or *failed*.

The process of aggregation and consistency determination depends largely on the semantics of special updates for a given background and its elements. Axioms of the internal structure of the elements guide the plug-in writer in determining what is considered consistent (i.e. what makes sense), and what is not. In some cases, the multiplicity of an update instruction performed on a given location is important in determining the semantics of the special update: [20] gives the example of the background class of counters to illustrate this point. For this reason, the data structure used for collecting update instructions is a *multiset* rather than a set.

The freedom given to plug-ins in determining their own aggregation promotes the extensibility of the engine with background classes for the widest possible variety of sorts.

3.1.8 Aggregation Phase Consistency

Once the aggregation phase is complete, aggregation consistency can be checked with the following function:

- **derived** *aggregationConsistent* : UpdateMultiset → Boolean
  returns *true* if aggregation was completed with consistency; *false* is returned otherwise. It is defined as:

$$aggregationConsistent(uMset) \equiv allUpdatesProcessed(uMset) \ \wedge noAggregationFailures(uMset)$$

There are two conditions which must be met in order to ensure the consistency of aggregation:

1. All updates in the multiset should have been processed (and should have some status flag). Every update instruction should have either been processed by the Basic Update Aggregator, or an aggregator plug-in.

derived *allUpdatesProcessed* : UpdateMultiset $\rightarrow$ Boolean
returns *true* if all update instructions have been processed; *false* is returned otherwise. It is defined as:

$$allUpdatesProcessed(uMset) \equiv$$
$$\forall ui \in uMset, \exists p \in \text{Plugin},\ aggregator(p)\ \wedge\ aggStatus(ui, p) \neq undef$$

2. There should be no update instructions in the multiset which have been flagged as *failed*.

   derived *noAggregationFailures* : UpdateMultiset $\rightarrow$ Boolean
   returns *true* if all locations were aggregated consistently; *false* is returned otherwise. It is defined as:

$$noAggregationFailures(uMset) \equiv$$
$$\forall ui \in uMset, \nexists p \in \text{Plugin},\ aggregator(p)\ \wedge\ aggStatus(ui, p) = failed$$

When the aggregation phase is considered to be inconsistent, this constitutes a failed step of the simulated machine (as does an inconsistent update set).

Notice that aggregation is not considered to be inconsistent if update instructions have been successfully processed more than once, potentially by multiple plug-ins. In such a situation, each plug-in processing instructions for a location will produce a resultant update for that location. This will not pose a problem if the two resultant updates do not conflict. However, if they do indeed conflict, this problem will be caught during the update set consistency check. Thus, multiple successfully processed update instructions are not always problematic and so a check for this situation is not incorporated into *aggregationConsistent* with the understanding that, if there is a problem, it will be caught during the update set consistency check.

3.1.9 Turbo ASMs and Sequential Composition

Aggregation as we have described it thus 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, this is insufficient. With the introduction of special updates resulting in the modification of elements at a given location, it is not always desirable for a Turbo ASM rule to return aggregated resultant updates.

In [24] the author discusses how support for sequential composition of update multisets is incorporated into the engine.

3.2 Parser

The parser module is used during the CoreASM specification loading step of the engine life cycle (Section 2.2) and performs two tasks:

1. Parse the specification header to determine the plug-ins required for a given specification.

2. Parse the entire specification.

3.2.1 The CoreASM Language Dependance on Specifications

During the initialization stage of the engine, depicted in Figure 2.4, the LoadStdPlugins rule loads all standard plugins; these plug-ins contain grammar extensions. While in the process of loading a specification (depicted in Figure 2.5) in the *Parsing Header* mode of the engine, the header of the specification is parsed via the ParseHeader rule of the Parser. The ParseHeader rule looks for **use** directives which specify additional plug-ins to be loaded for use in execution of the specification (see Figure 3.1).

```
// Header
use Tree
use Map

// Body
...
```

Figure 3.1: An example use of **use** directives in a CoreASM specification. While the standard plug-ins are automatically loaded, the `Tree` and `Map` plug-ins are loaded especially for use with this specification.

The engine mode then becomes *Loading Plug-ins*, where the loading of these additional plug-ins is done via the LoadSpecPlugins rule in the Control API. These additional plug-ins may also contain grammar extensions. Thus the CoreASM language syntax is dependent on the plug-in requirements of the specification to be interpreted, and hence can differ from one specification to another.

Once all plug-ins required by a specification have been loaded, the engine moves to the *Parsing Spec* mode of the engine, and executes the ParseSpecification rule:

Parser: Parse Specification

ParseSpecification ≡
  BuildGrammar
seq
  BuildAST

At a high-level, the act of parsing the specification can be broken down into two major steps:

1. Building the grammar to use based on all grammar extensions provided by loaded plug-ins.
2. Using this grammar to building the AST which represents the specification.

Here we concentrate on how the first step is achieved. The second step involves typical lexical and syntactic analysis which we do not describe here. We direct the reader to [1] for more information on these stages of interpretation.

3.2.2 Dynamic Grammar

Recall that the kernel contains only the minimum functionality necessary for basic DASM semantics. The kernel provides a grammar which specifies the syntax for

rules, operators, and literals to support these semantics. Regardless of which other plug-ins are required by a specification, the grammar included by the kernel is guaranteed to be present.

With this in mind, the *kernel grammar* has been structured in such a way that it is hierarchically segregated with respect to aspects of the language with extensible syntax.

```
Start -> ...
...
Rules -> ...
Operators -> ...
Literals -> ...
```

Figure 3.2: The kernel grammar structure showing GEPs.

In Figure 3.2 the general structure of the grammar is shown. The grammar is structured such that all productions describing rule form syntax are reached via the `Rules` production and all productions describing literal syntax can be reached via the `Literals` production. For each production extending the kernel grammar, we simply append that production to the RHS of the appropriate kernel grammar production. Productions which are designed to be extensible in this way are called *grammar extension points* (GEP).

For example, the Set Plug-In provides both new rule forms and literals. Assume that the productions which describe the Set Plug-In specific syntax are as follows:

```
...
SetRules -> ...
SetLiterals -> ...
```

Then the kernel grammar is extended at both the `Rule` and `Literal` GEPs with these additional productions:

```
Start -> ...
...
Rules -> ... | SetRules
Literals -> ... | SetLiterals
...
SetRules -> ...
SetLiterals -> ...
```

Extending the parser with new operators is a more complicated process that we describe here at a very high-level. In CoreASM the segment of the grammar which describes operator syntax is built dynamically with operator classes, precedence levels and associativity in mind. This operator precedence grammar is then integrated into the grammar via the `Operators` grammar extension point. Upon loading plug-ins, the engine is made aware of the operators provided by each plug-in, as well as operator class (i.e. unary, binary, ternary, etc.), operator associativity (i.e. LA or RA) and operator precedence (which in the case of CoreASM is specified

via a number between 0 and 100). Using all this information, the engine dynamically constructs the operator precedence grammar productions required to properly describe the syntax and characteristics of all operators to be supported.

3.3 Abstract Storage

Abstract storage models the state of the simulated machine in CoreASM. In addition, it also provides a set of practical operations affiliated to states. This section explores the architecture of this module and presents its interface to other components of the CoreASM engine.

3.3.1 Module Interface

Currently, it is not fully compatible with the Java interface.

Code Reference:

- `org.coreasm.engine.absstorage.AbstractStorage`

We model the simulated abstract state as a function $content : \textsc{State} \times \textsc{Loc} \rightarrow \textsc{Element}$, where locations are defined, as usual, by pairs of function names and arguments. With this assumption, the following functions and rules define the interface of the Abstract Storage to other components of the engine. All the functions in this section are *controlled* functions.

- $state : \textsc{State}$
  is the current state of the simulated machine.
- $getValue : \textsc{Loc} \rightarrow \textsc{Element}$
  returns the value of a given location. We have,
  $$getValue(l) = \begin{cases} content(state, l), & \text{if } content(state, l) \neq undef; \\ \mathsf{uu}, & \text{otherwise.} \end{cases}$$
- **rule** SetValue($l : \textsc{Loc}, v : \textsc{Element}$)
  sets the value of the given location to the given value. This rule is defined as follows:

  SetValue

  SetValue$(l, v) \equiv$
  $content(state, l) := v$

- **rule** PushState
  copies the current state in the stack. This rule is defined as follows[4]:

  Push

  PushState $\equiv$
  $asStack(asPtr) := state$
  $asPtr := asPtr + 1$

[4] We are assuming $asPtr = 0$ in the initial state.

- **rule** PopState
  retrieves the state from the top of the stack (thus discarding the current state). This routine is defined as follows:

Pop

```
PopState ≡
  state := asStack(asPtr − 1)
  asPtr := asPtr − 1
```

- **rule** Apply(u : Updates)
  applies the updates in the update set u to the current state.

- **rule** ClearState
  Clears the state to an empty state.

- *newElement* : Element
  returns a new element; i.e., imports a new element into the state and returns the imported element. This function is implemented by the following rule:

New Element

```
NewElement ≡
  return newValue(Element) in skip
```

- *newElementFrom* : UniverseElement → Element
  returns a new element of the given background or universe. If the argument is a background (see Section 3.3.3), it asks the given background to provide a new (perhaps default like 0 or "") element. Otherwise, it uses the *newElement* function (see above) to get a new element and adds it to the universe. This function can be implemented by the following rule:

New Element From

```
NewElementFrom(u) ≡
  return a in
    if u ∈ Background then
      a := newValue(u)
    else
      a := newElement
      uMember(u, a) := true
```

- *consistent* : Updates → Boolean
  holds if the update set is consistent according to [9, Def. 2.4.5].

- **rule** SetChosenAgent
  sets the value of *self* (see Section 2.2).

- **rule** GetChosenProgram
  loads the program of the current agent (*self*) from the state (see Section 2.2).

- **rule** FireUpdateSet
  fires the accumulated update set (see Section 2.2).

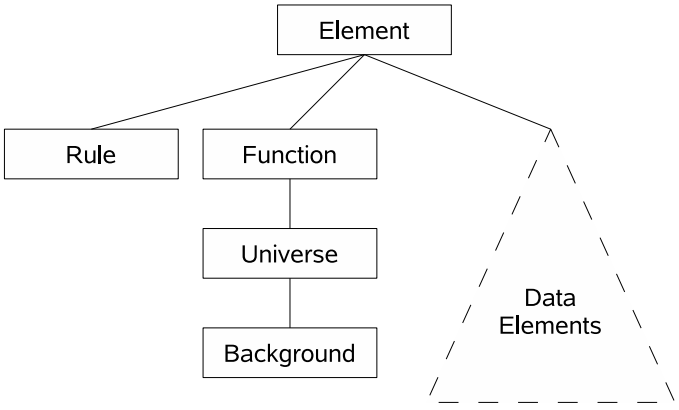

Figure 3.3: Core Elements in the Kernel

3.3.2 Elements of the State

Code Reference:

- `org.coreasm.engine.absstorage.Element`

The following functions are defined over all elements of the state.

- bkg : Element $\rightarrow$ Name
  is the name of the background of the given element. The default value is "Element".

- $equal_{Element}$: Element $\times$ Element $\rightarrow$ BOOLEAN
  returns *true* if the two elements are equal. We have

$$\forall a_1, a_2 \in \text{Element} \quad a_1 = a_2 \leftrightarrow equal_{Element}(a_1, a_2)$$

- **derived** $equal$: Element $\times$ Element $\rightarrow$ BOOLEAN
  returns *true* if the given elements are equal. This function is defined as

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

Element Enumerability

An enumerable element is any element which through some processing, can provide a collection of all the elements which constitute its internal structure. This general idea of enumerability can be easily applied to sets, a multisets, trees, records, etc.. The collection provided by these elements is a simple unordered group, which can contain duplicates.

The enumerable interface is useful as it provides a universal interface for all elements which can be represented (in albeit a simple form) by collection. The interface required by all enumerable elements is as follows:

- $enumerable$: Element $\rightarrow$ Boolean
  holds *true* if the element is enumerable, and *false* otherwise. The default value of this function is *false*.

- **derived** *enumerate* : Element → ElementCollection
  provides a collection of elements contained within the internal structure of the enumerable element, and is defined by its background:

$$enumerate(e) \equiv enumerate_{bkg(e)}(e)$$

Function Elements

Code Reference:

- `org.coreasm.engine.absstorage.FunctionElement`
- `org.coreasm.engine.absstorage.StateFunction`

FunctionElement extends Element to provide a core concept for state functions (tables) and custom-defined functions (e.g., functions provided by a plug-in, such as '$sin(x)$'). The following functions and rule are defined over Functions:

- *funcName* : FunctionElement → Name
  is the name of the function. If not *undef*, this name must be unique in state.
- *signature* : FunctionElement → Signature
  is the signature of the given function. The default value of this function is *undef*.
- *fGetValue* : FunctionElement × Element-seq → Element
  returns the value of this function with respect to the given arguments. The default value of this function is uu.
- **rule** FSetValue$(f, args, v)$
  sets a new value for the function, if this is possible. By default, this rule is defined as follows.

FSetValue

FSetValue$(f, args, v) \equiv$
  $fGetValue(f, args) := v$

- *fClass* : FunctionElement → FuncClass
  is the class of the function, where FuncClass is $\{monitored, controlled, out, static, derived\}$. the default value of this function is *controlled*.
- **derived** *fLocations* : FunctionElement → Loc-set
  if not *undef*, it is the set of all locations for which this function has a value other than *undef*.
- **derived** $equal_{Function}$: FunctionElement × Element → BOOLEAN
  where we have,

$$equal_{Function}(f_1, f_2) \equiv \forall a \in \text{Element-seq}\ fGetValue(f_1, a) = fGetValue(f_2, a)$$

- $\forall f \in$ FunctionElement $bkg(f) =$ "Function"

Universe Element

Code Reference:

- `org.coreasm.engine.absstorage.AbstractUniverse`
- `org.coreasm.engine.absstorage.UniverseElement`

UniverseElement extends FunctionElement by introducing the following functions:

- *uMember* : UniverseElement × Element → Boolean
  is the membership function of the universe. If u is a universe, then we may use the syntactical form $u(x)$ for $uMember(u, x)$.
- **derived** $equal_{Universe}$: UniverseElement × Element → BOOLEAN
  where
  $$equal_{Universe}(a, b) = equal_{Function}(a, b)$$
- $\forall i \in$ UniverseElement $bkg(i) =$ "Universe"

For all $v \in$ Element and $u \in$ UniverseElement, we have,

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

Rule Elements

Code Reference:

- `org.coreasm.engine.absstorage.RuleElement`

Rule extends Element by introducing the following function

- *ruleName* : Rule → Name
  is the name of the rule. If not *undef*, this name must be unique in state.
- *body* : Rule → Node
  holds the body (syntax tree) of the rule.
- *param* : Rule → Token-seq
  holds (in order) the parameters of the rule in squence of tokens (or strings).
- **derived** $equal_{Rule}$: Rule × Element → BOOLEAN
  where
  $$equal_{Rule}(a, b) = equal_{Element}(a, b)$$
- $\forall i \in$ Rule $bkg(i) =$ "Rule"

Boolean Elements

Code Reference:

- `org.coreasm.engine.absstorage.BooleanElement`

BooleanElement has only two elements, tt and ff, which represent boolean elements of *true* and *false*. This domain extends Element by introducing the following functions:

- **static** $valueOf_{Bool}$: Boolean $\rightarrow$ BooleanElement
  returns the Boolean element equivalence of the given Boolean value. It is defined as follows:
  $$valueOf_{Bool}(true) = \mathsf{tt}$$
  $$valueOf_{Bool}(false) = \mathsf{ff}$$
- **derived** *getBooleanValue* : BooleanElement $\rightarrow$ Boolean
  returns the Boolean value of this element.
- **derived** $equal_{Bool}$: BooleanElement $\times$ Element $\rightarrow$ Boolean
  where
  $$equal_{Bool}(a, b) \Leftrightarrow getValue_{Bool}(a) = getValue_{Bool}(b)$$
- $\forall b \in$ BooleanElement $\quad bkg(b) =$ "Bool"

Locations

Code Reference:

- `org.coreasm.engine.absstorage.Location`

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

- *locName* : Loc $\rightarrow$ Name
  is the name of the function on which this location is defined.
- *locArgs* : Loc $\rightarrow$ Element-seq
  is the list of abstract object values, as arguments of the location.
- **derived** *locFunction* : Loc $\rightarrow$ FunctionElement
  is the function on which this location is defined.
  $$locFunction(l) = f \Leftrightarrow \ name(f) = locName(l)$$

3.3.3 Background Element

Code Reference:

- `org.coreasm.engine.absstorage.AbstractUniverse`
- `org.coreasm.engine.absstorage.BackgroundElement`

Background extends UniverseElement by introducing the following additional function and restriction:

- *newValue* : Background → Element
  returns a pseudo new element of the given background; i.e., most probably returns a default value like an empty string for strings, or an empty set for sets. In the actual implementation, each background will implement its own version of this function. If this function returns *undef* it indicates that no new element can be created from this background. What is returned depends completely on view of a background; it may be seen in such a way that either elements have internal structure which can be modified (e.g. strings and sets) or not (e.g. integers, reals). A pseudo new element would be returned in the former case whereas *undef* would be returned in the latter.

- $\forall\ b \in$ Background $\ fClass(b) = static$
  It is not possible to change the membership function of a background; i.e., it is not possible to add any element to a background or to remove any element from it.

- **derived** $equal_{Background}$: Background × Element → BOOLEAN
  where
  $$equal_{Background}(a, b) = equal_{Universe}(a, b)$$

- $\forall i \in$ Background $\ bkg(i) =$ "Background"

3.3.4 State

Code Reference:

- `org.coreasm.engine.absstorage.State`
- `org.coreasm.engine.absstorage.TreeState`

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

- *content* : State × Loc → Element
  is the value of a given location in the state. This function represents the interface of the state.

- *universes* : State → UniverseElement-set
  is the set of all the defined universes in the state.

- *functions* : State → FunctionElement-set
  is the set of all the functions defined in the state.

- *rules* : State → Rule-set
  is the set of all the rules defined in the state.

- **derived** *locations* : State → Loc-set
  is the set of all defined locations (with a value other than *undef*) in the state. We have
  $$locations(s) = \{l \mid \exists f\ (f \in functions(s) \wedge l \in fLocations(f))\}$$

3.4 Scheduler

3.4.1 Module Interface

- The following rules directly contribute to one computation step of the engine (see Section 2.2 for more details):
  - **rule** InitializeState
    initializes the program state according to the specification. Rules and functions are created if they have been specified. Finally, the initial method of the specification is run.
  - **rule** StartStep
    starts a computation step.
  - **rule** RetrieveAgents
    gets the set of agents from the abstract storage.
  - **rule** SelectAgents
    selects a subset of agents contributing to the current computation step.
  - **rule** ChooseAgent
    chooses an agent from the selected agents set (see Section 2.2).
  - **rule** AccumulateUpdates
    accumulates the computed updates.
  - **rule** InitiateExecution
    clears the tree from previously computed values and points *pos* (the current position in the tree) to the root node of the tree that represents the current program (that is, the program of the current agent, as established above).
  - **rule** HandleFailedUpdate
    records that an inconsistent update was produced by executing the current step using the current selected agent set, so in future attempts to run this step, this selected agent set will not be considered by the scheduler.

- *updateInstructions* : Updates
  is the multiset of accumulating update instructions in one computation step.

- *updateSet* : UpdateSet
  is the set of computed updates in one computation step. This set is usually calculated at the end of each step based on the accumulated update instructions.

- *agentSet* : Agent-set
  is the set of all the available agents in the current state retrieved from the abstract storage at the beginning of every computation step.

- *selectedAgentSet* : Agent-set
  is the set of selected agents to contribute the computation of the current step.

- *chosenAgent* : Agent
  is the currently running (or to be running) agent.

- **monitored** *chosenProgram* : Program
  is the program of the chosenAgent. The value of this function is set by the abstract storage.

- *morePossibleSetsExist* : Boolean
  returns *true* if there exist more possible agent sets for execution of the current computation step; *false* is returned otherwise.

3.5 Interpreter

3.5.1 Module Interface

The following functions and rules define the interface of the Interpreter component to other components of the engine.

- **rule** ExecuteTree
  executes the parse tree addressed by *pos*, if *value*(*pos*) is not *undef*; otherwise it either switches the engine mode to *ChooseNextAgent* if *pos* refers to the top of the parse tree or moves *pos* one node up the parse tree. See Section 2.2 for more details.

3.5.2 Notation

We specify the interpreter as a collection of rules (some embedded in the kernel, others contributed by plug-ins) which traverse a parse tree while evaluating values, locations and updates. We state the following assumptions:

1. nodes in the tree are in the domain of the following (mostly partial) functions:
   - *first* : Node → Node, *next* : Node → Node, *parent* : Node → 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 go back to its parent, (see Figure 2.3 for reference);
   - *class* : Node → Class returns the syntactical class of a node (i.e., the name of the corresponding grammar non-terminal class);
   - *token* : Node → Token returns the syntactical token represented by the node (e.g., either a keyword, an identifier, or a literal value);
   - $[\![\cdot]\!]$: Node → Loc × Updates × Element holds the result of the interpretation 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)[5]. We access elements and establish properties of such triples through the following derived functions:
     - *loc* : Node → Loc returns the location (l-value) associated to the given node, i.e. $loc(n) \equiv [\![n]\!] \downarrow 1$.
     - *updates* : Node → Updates returns the updates associated to the given node, i.e. $updates(n) \equiv [\![n]\!] \downarrow 2$.

[5]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.

  - *value* : Node → Element returns the value (r-value) associated to the given node, i.e. $value(n) \equiv [\![n]\!] \downarrow 3$.
  - *evaluated* : Node → Boolean indicates if a node has been fully evaluated. We have,

$$evaluated(n) \equiv [\![n]\!] \neq undef$$

- *plugin* : Node → Plugin is the plug-in associated to expression and statement nodes, that is, the plug-in responsible for parsing and evaluation of the node.

2. a special variable *pos* holds at all times the current position in the tree;

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 $\square$ 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 set 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 set) 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 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

$$(\!|\, pattern \,|\!) \rightarrow \quad actions$$

     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 3.1; 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.

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

* 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 3.1: Abbreviations in syntactic pattern-matching rules.

Table 3.2 exemplifies how our compact notation can be translated into actual ASM rules.

4. the value of local variables (e.g., those defined in **let** rules) is maintained by a global dynamic function of the form env : Token → Element

5. a static function bkg : Element → Background provides, for any arbitrary value v, the background of the value or *undef* if the value is native in the core.

Notice that, according to the rule ExecuteTree previously described in Section 2.2, interpreter rules in the kernel or from plug-ins 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 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 3.2, 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.

3.5.3 Kernel Expression Interpreter

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

Literals are simply lifted to their semantic counterparts:

Compact notation	Actual rule
$(\!\vert\, \textbf{if}\ {}^{\alpha}\boxed{e}\ \textbf{then}\ {}^{\beta}\boxed{\gamma} \,\vert\!) \rightarrow\ pos := \alpha$	**if** $class(pos) \neq \mathsf{Id}$ $\wedge\ token(pos) = \mathsf{IfThen}$ $\wedge\ class(first(pos)) \neq \mathsf{Id}$ $\wedge\ \neg evaluated(first(pos))$ $\wedge\ class(next(first(pos))) \neq \mathsf{Id}$ $\wedge\ \neg evaluated(next(first(pos)))$ **then** $pos := first(pos)$
$(\!\vert\, \textbf{if}\ {}^{\alpha}v\ \textbf{then}\ {}^{\beta}\boxed{\gamma} \,\vert\!) \rightarrow\ \textbf{if}\ v = \mathsf{tt}\ \textbf{then}\ \ldots$	**if** $class(pos) \neq \mathsf{Id}$ $\wedge\ token(pos) = \mathsf{IfThen}$ $\wedge\ value(first(pos)) \neq undef$ $\wedge\ class(next(first(pos))) \neq \mathsf{Id}$ $\wedge\ \neg evaluated(next(first(pos)))$ **then** **if** $value(first(pos)) = \mathsf{tt}$ **then** ...
$(\!\vert\, \textbf{if}\ {}^{\alpha}v\ \textbf{then}\ {}^{\beta}u \,\vert\!) \rightarrow\ \ldots$	**if** $class(pos) \neq \mathsf{Id}$ $\wedge\ token(pos) = \mathsf{IfThen}$ $\wedge\ value(first(pos)) \neq undef$ $\wedge\ updates(next(first(pos))) \neq undef$ **then** ...

Table 3.2: Examples of how pattern matching notation is translated into ASM rules.

Kernel Expressions: Literals

$(\!\vert\, \textbf{true} \,\vert\!)$	$\rightarrow$	$[\![pos]\!] := (undef, undef, \mathsf{tt})$
$(\!\vert\, \textbf{false} \,\vert\!)$	$\rightarrow$	$[\![pos]\!] := (undef, undef, \mathsf{ff})$
$(\!\vert\, \textbf{undef} \,\vert\!)$	$\rightarrow$	$[\![pos]\!] := (undef, undef, \mathsf{uu})$

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 (which we will describe 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:

Kernel Expressions

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

$(\!|\, {}^{\alpha}x({}^{\lambda_1}\boxed{?}_1, \ldots, {}^{\lambda_n}\boxed{?}_n) \,|\!)$ →
if $isFunctionName(x)$ **then**
  choose $i \in [1..n]$ **with** $\neg evaluated(\lambda_i)$
    $pos := \lambda_i$
  ifnone
    let $l = (x, \langle value(\lambda_1), \ldots, value(\lambda_n)\rangle)$ **in**
      $[\![pos]\!] := (l, undef, getValue(l))$
if $undefined(x)$ **then**
  HandleUndefinedIdentifier$(x, \langle\lambda_1, \ldots, \lambda_n\rangle)$

where
$undefined(x) \equiv \nexists e \in$ Element : $name(e) = x$
$isFunctionName(x) \equiv \exists e \in$ Element : $name(e) = x \wedge isFunction(e)$

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 $[\![pos]\!]$.

Finally, if the interpreter encounters an identifier that is bound to no element in the state, the HandleUndefinedIdentifier rule will create a new function element with a default value of *undef* for the given arguments[6]:

HandleUndefinedIdentifier

HandleUndefinedIdentifier$(x, args) \equiv$
  choose $\lambda \in args$ **with** $\neg evaluated(\lambda)$
    $pos := \lambda$
  ifnone
    let $f = new($Element$)$ **do**
      $isFunction(f) := true$
      $name(f) := x$
      $[\![pos]\!] := ((x, args), undef, \mathsf{uu})$

Extending the standard definition, but in keeping with common practice, we also allow expressions to refer to functions (and rules, as we will see later), which can thus be treated as first-order objects in the language. The following rules apply to functions where the function itself is given as an expression. In these cases, we

[6]The definition of HandleUndefinedIdentifier presented here is for the "liberal" mode of CoreASM, which has no strict type checking and allows identifiers to be used without declaration, which is suited for early analysis and specification. In the "strict" mode, this macro is refined to cause an error.

first evaluate the expression, and if the result is a function value, we handle it as in the previous case. Notice though that we do not allow nullary functions to be accessed directly through an expression, to avoid syntactic ambiguity; in such cases, an empty pair of parenthesis has to be used to distinguish between the function value itself (without parenthesis) and the value of the nullary function represented by the function value (with parenthesis).

Kernel Expressions: Application

$(\!|\, {}^{\alpha}\square({}^{\lambda_1}?_1, \ldots, {}^{\lambda_n}?_n) \,|\!) \quad \rightarrow \quad pos := \alpha$

$(\!|\, {}^{\alpha}v({}^{\lambda_1}?_1, \ldots, {}^{\lambda_n}?_n) \,|\!) \quad \rightarrow$

if *isFunction*(v) **then**
  choose $i \in [1..n]$ **with** $\neg$*evaluated*(λ_i)
    $pos := \lambda_i$
  ifnone
    let x = *name*(v) **in**
      let $l = (x, \langle value(\lambda_1), \ldots, value(\lambda_n)\rangle)$ **in**
        $[\![pos]\!] := (l, undef, getValue(l))$

3.5.4 Kernel Rule Interpreter

Rule plug-ins provide the semantics of executing of rules. Execution of rules results in a set of update instructions that is the underlying value for the rule node of the parse tree. As discussed in Section 2.2, 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.

To evaluate an identifier as a rule, the interpreter first checks if a rule element is bound the identifier. If so, the RuleCall macro is called to execute the rule, which we will describe shortly. Notice that in this case arguments are *not* evaluated prior to calling the rule: in fact, the semantics of rule calls in [9] prescribes that the entire term used as actual argument must be substituted to the formal parameter in the body of the rule, not its value. Also, note that when the rule to call is denoted through an expression, the rule $(\!|\, {}^{\alpha}\square({}^{\lambda_1}?_1, \ldots, {}^{\lambda_n}?_n) \,|\!) \rightarrow \ pos := \alpha$ from functional application above applies, hence we do not need to repeat it here; after evaluation, the pattern $v(?_1, \ldots, ?_n)$ applies, for which we provide here another rule (mutually exclusive with the one for functional application) to handle rule calls.

Kernel Rules

$(\!|\, {}^{\alpha}x \,|\!) \quad \rightarrow$

if *isRuleName*(x) **then**
  RuleCall(*ruleValue*(x), $\langle\rangle$)

$(\!|\, {}^{\alpha}x({}^{\lambda_1}?_1, \ldots, {}^{\lambda_n}?_n) \,|\!) \quad \rightarrow$

if *isRuleName*(x) **then**
  RuleCall(*ruleValue*(x), $\langle\lambda_1, \ldots, \lambda_n\rangle$)

$(\!|\, {}^{\alpha}v({}^{\lambda_1}?_1, \ldots, {}^{\lambda_n}?_n) \,|\!) \quad \rightarrow$

if *isRule*(v) **then**
  RuleCall($v, \langle\lambda_1, \ldots, \lambda_n\rangle$)

where
isRuleName(x) $\equiv \exists e \in$ Element : *name*(e) = $x \wedge$ *isRule*(e)

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 parameters 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 [9] Section 4.1.2. Here, we model macro calls, while the effect of submachine calls can simply be achieved by using 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 [9], Definition 2.4.18). For simplicity, we do not explicitly check for such conditions in our specification.

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

RuleCall

```
RuleCall(r, args) ≡
  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.

CopyTreeSub

```
CopyTreeSub(α, ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩) ≡
  return result in
    if α ≠ undef then
      if class(α) = Id ∧ ∃i s.t. token(α) = x_i then
        result := CopyTree(λ_i)
      else
        let n = new(Node) in
          first(n) := CopyTreeSub(first(α), ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩)
          next(n) := CopyTreeSub(next(α), ⟨x_1, ..., x_n⟩, ⟨λ_1, ..., λ_n⟩)
          class(n) := class(α)
          token(n) := token(α)
          plugin(n) := plugin(α)
          result := n
    else
      result := undef
```

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

Kernel Rules: Assignment

```
(| α[?] := β[?] |)  →  choose τ ∈ {α, β} with ¬evaluated(τ)
                         pos := τ
                       ifnone
                         if loc(α) ≠ undef
                           [[pos]] := (undef, {|⟨loc(α), value(β)⟩|}, undef)
                         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 plug-in can contribute new forms of expressions which, as long as they result in a location, are deemed syntactically acceptable in the lhs of an assignment.

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

KernelInterpreter: import

```
(| import α x do β[r] |)  →  let e = new(Element) in
                               env(x) := e
                             pos := β

(| import α x do β u |)   →  env(x) := undef   // No nesting
                             [[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 r is then evaluated in this new environment by assigning *pos* to the corresponding node. The local value of the given identifier is then set to *undef* when the evaluation of the rule part is complete.

3.5.5 Operators

Plug-ins can extend the CoreASM language by introducing new expression forms. One of the most important form of such extension is adding new operators (unary or binary) to the languge. To avoid lengthy expressions with unnecessary paranthesis, the engine provide plug-ins with a mechanism to declare a precedence level for their operators.

Precendence level of an unary or binary operator is defined by a numeric value $p \in [0 \dots 10]$, 10 being the highest. This value should be attached to all patterns that define a new expression form based on an operator. The following example introduces a new operator Ω with precedence level 3:

$$\langle\!| \boxed{e} \ \Omega \ \boxed{e} |\!\rangle_{[3]} \rightarrow \quad \dots$$

An in-depth discussion of operator evaluation in CoreASM is presented in [24].

3.6 The Plug-in Framework

Plug-ins can extend the engine by

- adding new grammar rules to the Parser,
- adding new semantic rules to the Interpreter,
- adding new backgrounds and operations to the Abstract Storage, and
- adding new policies to the scheduler. how?

In addition to these basic extensions, plug-ins can also register themselves for a *Plug-in Point*. A set is assigned to every plug-in point, and the plug-ins register for this points by adding themselves to the appropriate set. At any plug-in point in the execution cycle of the engine, if there is any plug-in registered for that point, the appropriate plug-in rule is executed before the engine continues its execution.

Plug-in Points are transitions of the engine mode from one mode to another. The execution of plugins registered for these mode transitions is handled in the Next routine. We have,

Next

```
Next(newMode : EngineMode) ≡
  seq
    forall p ∈ registeredPlugins(engineMode, newMode)
      FirePluginOnModeTransition(p, engineMode, newMode)
    engineMode := newMode
```

To emphasize that calling plugins will occure in a sequential manner, we can further refine Next to the following rule:

Next: Refined

```
Next(newMode : EngineMode) ≡
  MarkPlugins(newMode)
  seq FirePlugins(newMode)
  seq engineMode := newMode
```

where the auxiliary routines are defined as follows:

derived *registeredPlugins* : EngineMode × EngineMode → Plugin-set
registeredPlugins(*src*, *trg*) =
{*p* | *p* ∈ Plugin ∧ (*src* ∈ *pluginSourceModes*(*p*) ∨ *trg* ∈ *pluginTargetModes*(*p*))}

MarkPlugins(*newMode* : EngineMode) ≡
forall *p* ∈ *registeredPlugins*(*engineMode*, *newMode*)
fireFlag(*p*) := *true*

FirePlugins(*newMode* : EngineMode) ≡
iterate
choose *p* ∈ *registeredPlugins*(*engineMode*, *newMode*) **with** *fireFlag*(*p*)
FirePluginOnModeTransition(*p*, *engineMode*, *newMode*)
fireFlag(*p*) := *false*

All the Plug-ins are loaded as the first phase of parsing. There are other possible options that are not discussed in this paper (see [Issue 108]). Plug-ins that need to be loaded are listed in the top most section of CoreASM specifications. In *Parsing Header* state (see Figure 2.5), the parser will use the interpreter to load the plug-ins. Plug-ins are listed using the '**use** plugin-name' syntaxt. In the following rule, The LoadPlugin rule finds a plugin with the given name, loads the plugin, and intializes it by calling the InitializePlugin rule defined later in this section.

Loading Plugin

⟪ **use** *x* ⟫ → LoadPlugin(*x*)

3.6.1 Plug-in Background

Code Reference:

- `org.coreasm.engine.Plugin`

Plugin is the background of all plug-ins that extend the functionality of the CoreASM engine. The following functions are defined over plugins:

- *pluginName* : Plugin → Name
  is the unique name of a plug-in. The engine cannot load two plug-ins that share the same name. In practice, this name could be the name of the Java class that implements the plug-in.
- *pluginRule* : Plugin → Rule
  returns the main rule of the plug-in which is used by by the interpreter (see routine ExecuteTree in Section 2.2).
- *pluginGrammar* : Plugin → Grammar
  returns the grammar (a set of grammar rules) provided by this plug-in.
- *pluginPolicy* : Plugin → Rule
  returns the scheduling policy provided by this plug-in.

- *pluginSourceModes* : Plugin → EngineMode-set
  is a set of engine modes; upon transition of the engine mode from any of these modes, the given plug-in must be notified (see routine Next).

- *pluginTargetModes* : Plugin → EngineMode-set
  is a set of engine modes; upon transition of the engine mode to any of these modes, the given plug-in must be notified (see routine Next).

- **rule** InitializePlugin(Plugin)
  initializes the plug-in. This rule is called for all the plug-ins that are loaded by the engine.

- **rule** FirePluginOnModeTransition(p : Plugin, *fromMode* : EngineMode, *toMode* : EngineMode) is called when the engine mode is switched from a mode or to a mode that this plug-in is registered for.

A background plug-in can also provide special update instructions and aggregation service for those instructions. Such a plug-in is called an *aggregator*, and the following functions and rules are defined for aggregator plug-ins:

- *isAggregator* : Plugin → Boolean
  holds if the plugin is an aggregator plugin.

- *updateInstructions* : Plugin → Action-set
  returns the set of special update instructions associated with this plugin. [7]

- *aggregateUpdates* : Plugin × Updates → Updates
  aggregates special update instructions that this plugin is responsible for and returns the aggregated set of updates.

- *compose* : Plugin × Updates × Updates → Updates
  composes the update multi-sets assuming that there is an order between them (first happens before second) and returns a multi-set of update instructions as the result of composition. Only instructions and locations for which the plug-in is responsible for composing are processed. Some major differences between $compose(p, u_1, u_2)$ and $aggregateUpdates(p, u)$ are:

  1. *aggregateUpdates* works with the current state and the update multiset provided to produce resultant updates; these resultant updates are generated to (when the update set is fired) replace values at locations in the state. However *compose* works exclusively with the update multisets provided, to produce a composed update multi-set which will *later* be aggregated (possibly with additional update instructions).
  2. The result of *aggregateUpdates* is a set of regular updates, while the result of *compose* is a multi-set possibly containing special update instructions.
  3. *compose* combines two multi-set of update instructions considering that there is an order between the occurance of the update instructions within the two multi-sets.

[7] *updateActions* : Plugin → Action-set
returns the set of special update actions associated with this plugin.

Chapter 4

Final Remarks

4.1 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` [19, 21] and, even further back, to the Prolog-based interpreter by Angelica Kappel [23]. Other interpreters and compilers followed: the lean *EA* compiler [4] from Karlsruhe University, the *scheme*-interpreter [14] from Oslo University, and an experimental EA-to-C++ compiler developed at Paderborn University [3]. Besides practical work on ASM tools, conceptual frameworks for more systematic implementations were developed. The work on the *evolving algebra abstract machine (EAM)* [12], 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.

Based on such experience, a second generation of more mature ASM tools and tool environments was developed: *AsmL* (ASM Language) [25], the *ASM Workbench* [11] and the *Xasm* (Extensible ASM) language [2] are all based on compilers, while *AsmGofer* [27] provides an ASM interpreter.[1] The most prominent one is AsmL, developed by the Foundations of Software Engineering group at Microsoft Research. 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 fletched programming language. At the same time, it lacks any built-in support for dealing with distributed systems. Being deeply integrated with the software development, documentation, and runtime environments of Microsoft, its design was shaped by practical needs of dealing with fairly complex requirements and design specifications for the purpose of software testing; as such, it is oriented toward the world of

[1] We focus here on the more common and well-known ASM tools. For a complete overview, see also [9], Sect. 8.3.

code. This has made it less suitable for initial modeling at the peak of the problem space and also restricts the freedom of experimentation.

The *ASM Workbench* is a tool environment supporting software specification, design, and validation in early design phases and rapid prototyping of embedded systems [13, 10]. The source language for the ASM Workbench tools is the *ASM Specification Language* (ASM-SL), a strongly typed language with an ML-like type system based on parametric polymorphism. ASM-SL extends the basic language of ASM transition rules by introducing additional constructs for defining ASM states, including a collection of predefined generic data types implementing standard mathematical structures (like tuples, lists, finite sets, finite maps, etc.) with associated operations. The ASM-SL language is quite concise and close to standard mathematical notation, making it easily readable and understandable. ASM-SL does however not provide any built-in support for distributed ASM models. In [26], a compilation scheme for compiling ASM-SL like specifications to C++ is presented, providing efficient C++ coding while preserving the structure of the original ASM specification. Based on this work, a proprietary compiler was developed and used successfully in the FALKO project at Siemens, Munich [7].

Xasm is an open source project [3] and comes with a development environment consisting of an Xasm-to-C compiler, a run-time system and a graphical interface for debugging and animating Xasm models. The language provides an interface to C allowing both C-functions to be used in Xasm programs as well as Xasm modules to be called from within C-programs. A rapid prototyping tool *Gem-Mex*, built around Xasm, assists the designer of a programming language in a number of activities related to the language life cycle (from early design steps to routine programmer usage). Gem-Mex supports automatic generation of documentations, generation of language implementations based on Xasm code, and visualization and animation of the static and dynamic behavior of specified languages at a symbolic level. Xasm in its present form does not support distributed ASMs.

Finally, AsmGofer is an advanced ASM programming system which runs on various platforms, including Unix-based or MS-based operating systems. It provides an ASM interpreter embedded in the functional programming language Gofer, a subset of Haskell, the de-facto standard for strongly typed lazy functional programming languages. A widely recognized application of AsmGofer is its use for executing the ASM specification of a light control system [8]. As with AsmL, ASM-SL and Xasm, AsmGofer does also not provide built-in support for distributed ASM models.

In contrast to CoreASM, all the above languages build on predefined type concepts rather than the untyped language underlying the theoretical model of ASMs; none of these languages comes with a run-time system supporting the execution of distributed ASM models; only Xasm is designed for systematic language extensions and in that respect is similar to our approach; however, the Xasm language itself diverts from the original definition of ASMs and seems closer to a programming language.

4.2 Conclusion

In this report we presented the design of the CoreASM extensible ASM execution engine, mainly focusing on the kernel of the engine. The CoreASM engine itself forms the kernel of a novel environment for model-based engineering of abstract

requirements and design specifications in the early phases of the software development process. Sensible instruments and tools for writing an initial specification call for maximal flexibility and minimal encoding as a prerequisite for easy modifiability of formal specifications, as required in evolutionary modeling for the purpose of exploring the problem space. The aim of the CoreASM effort is to address this need for abstractly executable specifications.

Aiming at a most flexible and easily extensible CoreASM language, most functionalities of the CoreASM engine are implemented through plug-ins to the basic CoreASM kernel. The architecture supports plug-ins for backgrounds, rules and scheduling policies, thus providing extensibility in three different dimensions. Hence, CoreASM adequately supports the need to customize the language for specific application contexts, making it possible to write concise and understandable specifications with minimal effort.

The CoreASM language and tool architecture for high-level design, experimental validation and formal verification of abstract system models is meant to complement other existing approaches like AsmL and XASM rather than replacing them. As part of future work, we envision an interoperability layer through which abstract specifications developed in CoreASM can be exported, after adequate refinement, to AsmL or XASM for further development.

Appendix A

Rules and Definitions

A.1 Engine Life-cycle

Initializing the Engine

InitKernel ≡
  pluginCatalog := {}
  loadedPlugins := {}
  grammarRules := {}

LoadCatalog ≡
  forall *pName* **in** *availablePluginModules* **do**
    let *p* = *createPlugin*(*pName*) **in**
      add *p* **to** *pluginCatalog*

LoadStdPlugins ≡
  forall *p* **in** *stdPlugins* **do**
    LoadPlugin(*p*)

LoadPlugin(*p*) ≡
  InitializePlugin(*p*)
  add *p* **to** *loadedPlugins*
  forall *r* **in** *pluginGrammar*(*p*) **do**
    add *r* **to** *grammarRules*
  forall *op* **in** *pluginOperators*(*p*) **do**
    add *r* **to** *operatorRules*

Loading Specification

```
ParseHeader ≡
  requiredPlugins := requestedPlugins(specification)

LoadSpecPlugins ≡
  forall p in requiredPlugins do
    LoadPlugin(p)

InitAbstractStorage ≡
  CreateFunction("agents")
  CreateFunction("program")
  CreateFunction("self")

ExecuteInitialization ≡
  CreateSpecFunctionsAndRules(specification)
  PrepareInitialRuleExecution

CreateFunction(name) ≡
  if isFunctionName(name) then
    ClearFunction(name)
  else
    let f = new(Element) in
      isFunction(f) := true
      name(f) := name

PrepareInitialRuleExecution ≡
  let a = new(Element) in
    SetValue(("agents", ⟨a⟩), tt)
    SetValue(("program", ⟨a⟩), initRule)
```

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[10] G. Del Castillo and U. Glässer. Simulation and validation of high-level Abstract State Machine specifications. In H. Szczerbicka, editor, *Modelling and Simulation: A Tool for the Next Millenium – Proc. of the 13th European Simulation Multiconference*, volume 2, pages 463–465, June 1999.

[11] 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.

[12] G. Del Castillo, I. Durdanović, 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.

[13] G. Del Castillo and U. Glässer. Computer-Aided Analysis and Validation of Heterogeneous System Specifications. In F. Pichler, R. Moreno-Diaz, and P. Kopacek, editors, *Computer Aided Systems Theory: Proceedings of the 7th International Workshop on Computer Aided Systems Theory (EUROCAST'99)*, volume 1798 of *LNCS*, pages 55–79. Springer, 2000.

[14] 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.

[15] R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An Extensible ASM Execution Engine. In *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

[16] 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.

[17] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[18] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[19] 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.

[20] Y. Gurevich and N. Tillmann. Partial Updates: Exploration. *Journal of Universal Computer Science*, 7(11):917–951, 2001.

[21] J. Huggins. An offline partial evaluator for evolving algebras. Technical Report CSE-TR-229-95, University of Michigan, 1995.

[22] ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition.* International Telecommunication Union, 2001.

[23] 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.

[24] Mashaal A. Memon. Specification language design concepts: Aggregation and extensibility in coreasm. Master's thesis, Simon Fraser University, Burnaby, Canada, Spring 2006.

[25] Microsoft FSE Group. *The Abstract State Machine Language.* Last visited March 2006, `http://research.microsoft.com/fse/asml/`.

[26] J. Schmid. Compiling Abstract State Machines to C++. *Journal of Universal Computer Science*, 7(11):1068–1087, 2001.

[27] Joachim Schmid. *Executing ASM Specitications with AsmGofer.* Last visited March 2006, `http://www.tydo.de/AsmGofer/`.

[28] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.