Fundamenta Informaticae 77 (2007) 71–103
IOS Press

CoreASM: An Extensible ASM Execution Engine*

Roozbeh Farahbod†
School of Computing Science, Simon Fraser University
Burnaby, B.C., Canada
rfarahbo@cs.sfu.ca

Vincenzo Gervasi
Dipartimento di Informatica, Università di Pisa,
Pisa, Italy
gervasi@di.unipi.it

Uwe Glässer
School of Computing Science, Simon Fraser University
Burnaby, B.C., Canada
glasser@cs.sfu.ca

Abstract. In this paper we introduce a new research effort in making *abstract state machines* (ASMs) executable. The aim is to specify and implement an execution engine for a language that is as close as possible to the mathematical definition of pure ASMs. The paper presents the general architecture of the engine, together with a high-level description of the extensibility mechanisms that are used by the engine to accommodate arbitrary backgrounds, scheduling policies, and new rule forms.

Keywords: CoreASM, Abstract state machines, Specification languages, Executable specification

1. Introduction

Abstract state machines [16], or ASMs, are well known for their versatility in computational and mathematical modeling of architectures, languages, protocols and virtually all kinds of sequential, parallel and

*This paper is a revised and updated version of [26].
†Address for correspondence: School of Computing Science, Simon Fraser University, Burnaby, B.C., Canada

distributed systems with an orientation towards practical applications. The particular strength of this approach is the flexibility and universality it provides as a mathematical framework for semantic modeling of functional requirements in terms of abstract machine models and their runs. Building on a rigorous mathematical foundation [5, 37], ASM abstraction principles provide an effective instrument for modeling the construction of software designs prior to coding and for analyzing such designs by reasoning about design choices and their implications. Typical deficiencies often hidden in informal requirements, such as ambiguities, loose ends and inconsistencies, thereby become explicit and can more easily be eliminated. To this end, abstraction and formalization help gaining a clearer understanding of the problem to be solved, thus reducing the risk of making premature decisions with fatal consequences [41].

Viewing behavior of discrete dynamic systems as evolution of abstract states, formally represented as variants of Tarski structures, is invaluable for bridging the gap between informal requirements and precise specifications in the earlier phases of system design. Similarly, this angle also simplifies the task of constructing models of requirements that are being extracted from implementations in reverse engineering applications. Both directions have been studied extensively by ASM researchers and developers in academia and industry for more than 15 years, leading to a solid methodological foundation for building *ASM ground models* [9]. Intuitively, an ASM ground model may be considered a semantic 'blueprint' of the key system requirements that need to be established in a precise and reliable form without compromising any conceivable refinements [10]. The role and nature of ground models, as discussed in [9], leads itself to the conclusion that the concept of ground model is inevitably present in every system design, but often not in an explicit form. Thus, the origin and motivation for the development of ASM specification, validation and verification techniques [8] has been the desire to make ground models visible and inspectable by analytical means and empirical techniques, exploiting machine assistance where appropriate. Widely recognized applications include semantic foundations of industrial system design languages like the ITU-T standard for SDL [33, 48, 25, 43], the IEEE language VHDL [13, 12] and its successor SystemC [47], programming languages like JAVA [51, 23], C# [11] and Prolog [6, 7], Web service description languages [29, 30, 28], communication architectures[34, 35], embedded control systems [15, 4, 14], et cetera.[1]

The research we describe in this paper 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 machine models. The CoreASM environment consists of 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, called Control API, thereby enabling future development and integration of complementary tools, e.g., for symbolic model checking [21] and automated test generation [32]. The design of CoreASM is novel and the underlying principles are unprecedented among the existing executable ASM languages, including the most advanced ones: AsmL [46], the ASM Workbench [18], XASM [1], and AsmGofer [49].

Exploring the problem space for the purpose of writing an *initial specification* calls for a language that emphasizes freedom of experimentation and supports the evolutionary nature of design being a creative activity. Such a language must allow writing highly abstract and concise specifications by minimizing the need for encoding in mapping the problem space to a formal model. The principle of minimality, in combination with robustness of the underlying mathematical framework, also makes easy modifiabil-

[1] See also the ASM website at *www.eecs.umich.edu/gasm/* and the overview in [16].

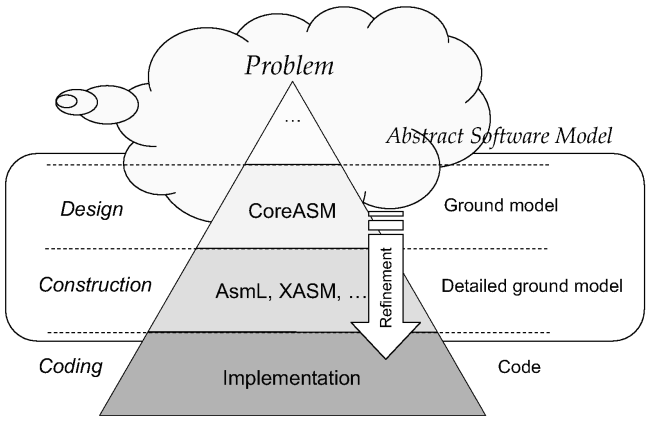

Figure 1. Background and Motivation.

ity feasible, effectively supporting the highly iterative nature of specification and design. 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.

The CoreASM language and supporting tool architecture focus on early phases of the software design process; consequently, primary concerns are towards 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 technique for specification with refinement that has been exploited in [16] and [10]. In this process, we aim at maintaining executability of even fairly abstract models. Another important characteristic that differentiates our endeavor from previous experiences is the emphasis that we are placing on extensibility of the language. Historical developments have shown how the original, basic definition of ASMs from the Lipari Guide [36] 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[2], often with non-standard operations. We want to preserve in our language the freedom of experimentation that has proven so fruitful in the development of ASM concepts, and, to this end, we have designed our architecture around the concept of *plug-in*s that allows to customize the language to specific needs. The argumentative structure leading from our high-level goals to specific design choices is summarized graphically in Figure 2.

An extensible, platform independent tool package (the language, its engine, 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 researchers that will be able to test in practice proposed extensions to the basic ASM language.

This paper is structured as follows. Section 2 provides first a high-level overview of the architecture of the CoreASM engine, and then presents its components in some detail; a discussion of the extensibility provisions in the architecture completes the section. Section 3 presents an abstract specification of the CoreASM language, and shows, through several examples, how the core language and its extensions are

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

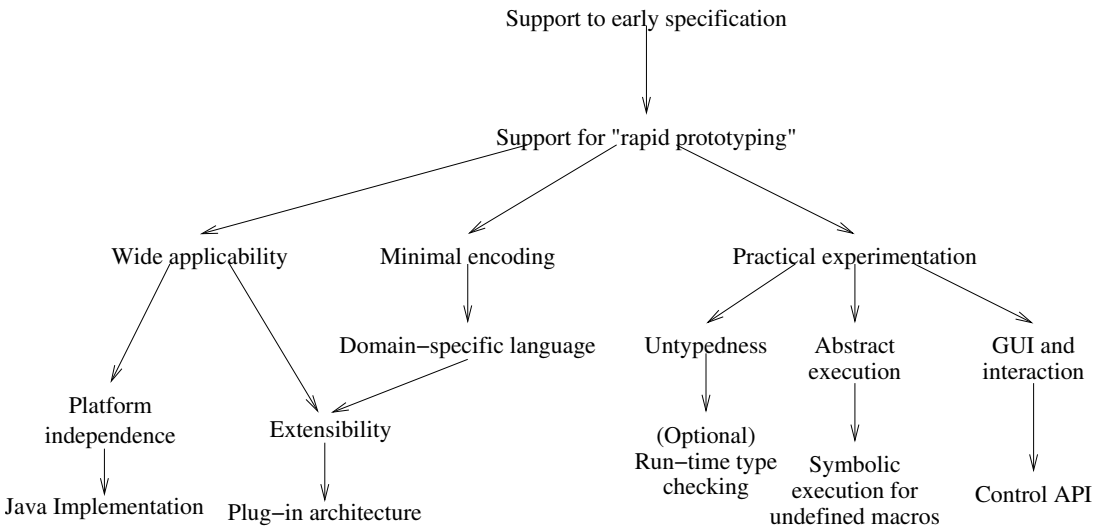

Figure 2. CoreASM requirements and design choices.

specified. Section 4 provides an account of related work; this is followed by our conclusions and plans for future work, which conclude the paper.

2. Architecture Overview

The CoreASM engine consists of four components: a *parser*, an *interpreter*, a *scheduler*, and an *abstract storage* (Figure 3). 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.

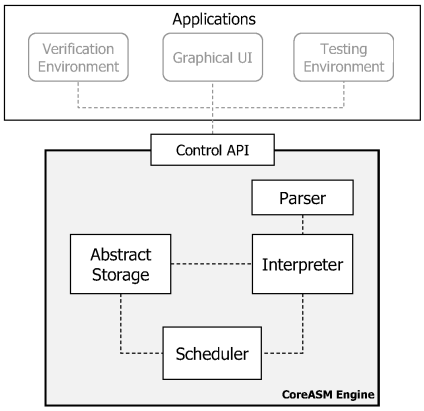

Figure 3. Overall Architecture of CoreASM.

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.[3] 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 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 6 to 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.[4]
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 4). 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,

[3] It is important to mention that the rollback mechanism can only rollback a simulated environment as part of a simulated run (e.g., when monitored function values are read from a file), whereas the "real" environment (e.g., a now function reading a real-time clock) cannot be rolled back.

[4] 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.

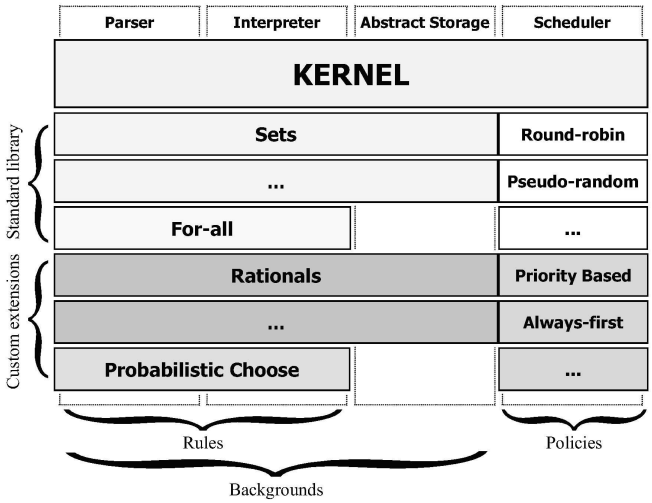

Figure 4. Layers and Modules of the CoreASM Engine.

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 are instances of 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 spirit in which successive layers of the Java [51] and C# [11] 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 5, 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 82)[5]. 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.

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

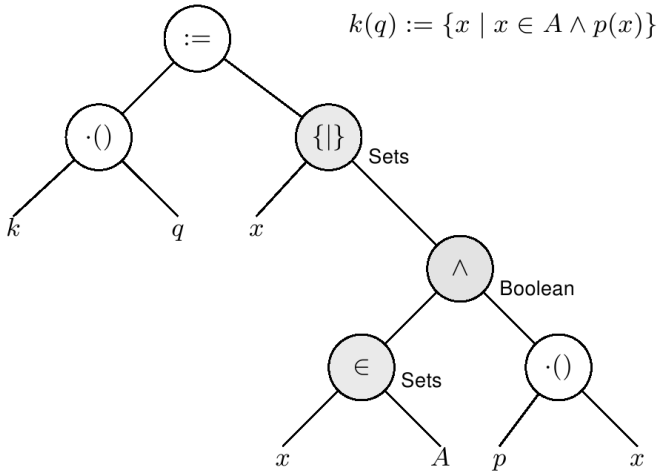

Figure 5. Sample Annotated Parse Tree.

In addition, it also provides other auxiliary information about the locations of current state, such as the ranges and domains of functions[6] 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 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 [16], 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
   (a) Initializing the kernel
   (b) Loading the plug-ins library catalogue
   (c) Loading and activating plug-ins from a standard library

[6]Here, by *range* and *domain* of a function we respectively refer to the set of all arguments for which the function value is not *undef* and the set of all function values which are not *undef*.

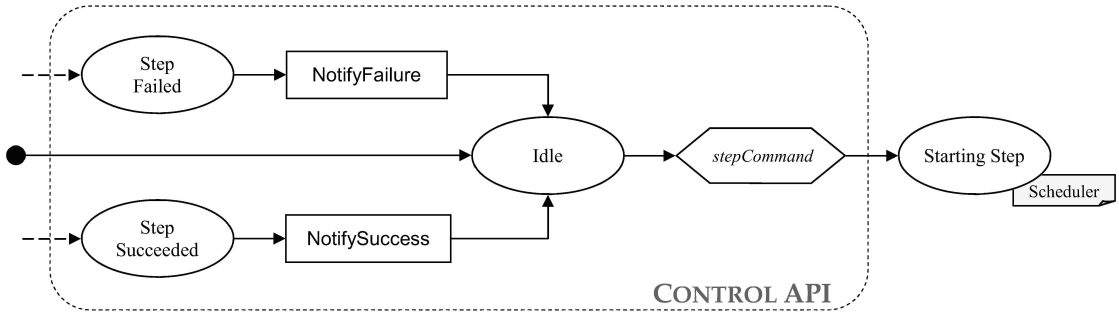

Figure 6. Control State ASM of a STEP command: Control API Module.

2. Loading a CoreASM specification
   (a) Parsing the specification header
   (b) Loading further needed plug-ins as declared in the header
   (c) Parsing the specification body
   (d) Initializing the abstract storage
   (e) Setting up the initial state
3. Execution of the specification
   (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).

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 6 to 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 6). In this state, the engine simply waits for a *STEP* command from the environment[7] (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.

[7]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.

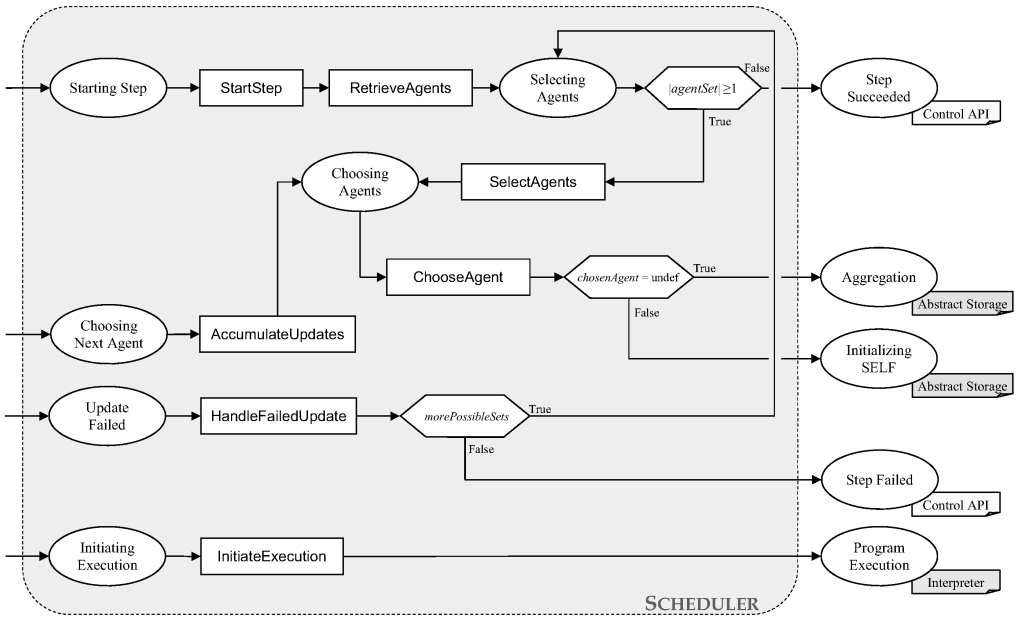

Figure 7. Control State ASM of a STEP command : Scheduler.

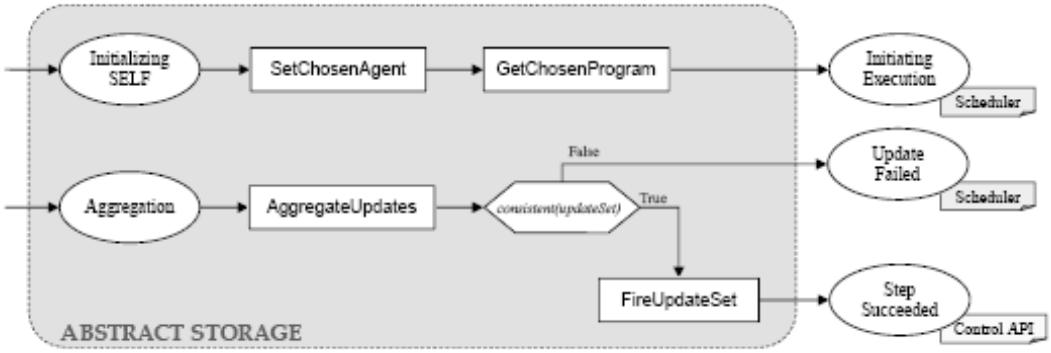

Figure 8. Control State ASM of a STEP command : Abstract Storage.

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 (a dual macro SetValue models the process of sending a $(l, value)$ pair to the abstract storage module for storing). 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*.

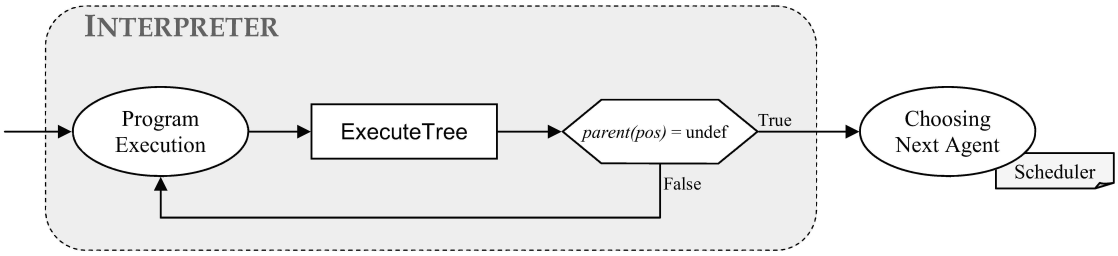

Figure 9. Control State ASM of a STEP command : Interpreter.

Scheduler

StartStep $\equiv$
  updateInstructions $:= \{\}$
  agentSet $:=$ *undef*
  selectedAgentsSet $:= \{\}$

RetrieveAgents $\equiv$
  agentSet $:=$ *getValue*((“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. Passing then through the *Choosing Agents* state, 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 Agents* until all selected agents have been executed.

Scheduler

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

ChooseAgent $\equiv$
  choose a **in** *selectedAgentsSet* **do**
    remove a **from** *selectedAgentsSet*
    chosenAgent $:= a$
  ifnone
    chosenAgent $:=$ *undef*

AccumulateUpdates $\equiv$
  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 $\equiv$
  SetValue((“self”, $\langle\rangle$), *chosenAgent*)

GetChosenProgram $\equiv$
  chosenProgram := *getValue*((“program”, $\langle$“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 [51], 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, the InitiateExecution rule removes the previously computed values from the tree (through the ClearTree macro, and sets the current position in the tree (denoted by the nullary function pos) 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 $\equiv$
  pos := *root*(*chosenProgram*)
  ClearTree(*pos*)

The specification of the interpreter is explored in more detail in Section 3. 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) 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 3.1 presents a more precise definition of these functions.

Interpreter

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

Notice also in the macro above how as soon as a node is fully evaluated (last **else** branch), the current position *value*(*pos*) is moved back to the parent node, if any, to continue evaluation.

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 Agents* 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 of the simulated ASM by FireUpdateSet (thus obtaining its next state), or provides an indication of failure by changing the state of the CoreASM engine to *Update Failed*. It is worthwhile to remark that aggregation, which is the process of interpreting accumulated update instructions to generate a set of updates, is obtained by delegating the actual interpretation to those plug-ins that provide aggregation services, as shown in the rule below:

Abstract Storage

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

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

Update instructions (vs. basic ASM updates) and the aggregation phase that aggregates those instructions into basic ASM updates are designed to support simultaneous incremental modification of data structures in CoreASM. The idea of update instructions is inspired by the work of Gurevich and Tillmann on partial updates [39, 40] where they provide an algebraic framework to support simultaneous partial modification of data structures in Parallel ASMs and a systematic approach to ensure the consistency and integrity of such modifications. The relationship between the idea of update instructions in

CoreASM and the partial update framework deserves an in-depth comparison of the two approaches and a discussion on their pros and cons, which is beyond the scope of this paper. We intend to further address this in a separate paper.

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[8]. It is worthwhile to note that only a single scheduling policy can be in force at any given time, whereas an arbitrary number of background and rule plug-ins can be all in use at the same time.

In CoreASM, the kernel (see Figure 4) 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

[8]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.

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 5), 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 a single agent at a time, which is sufficient for multi-agent ASMs where no assumptions are made on the scheduling policy[9].

In addition to modular extensions of the engine, plug-ins can also register themselves for *Extension Points*. Each state 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 state. 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 [16]), 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.

3. The CoreASM Language

We specify the CoreASM language (both its syntax and the corresponding semantics) through the specification of an interpreter. The specification of the kernel interpreter presents the core constructs of the language, which is then extended by standard library plug-ins to include basic ASM constructs such as the block rule and the conditional rule. The language is then further extended by Turbo ASM rules such as the sequence rule (**seq**) and the iterator (**iterate**). An example of extending the language with non-standard rules is also provided at the end of this section. For a comprehensive specification of the

[9]Notice that this particular scheduling policy guarantees the coherence condition of partially ordered runs, which is not guaranteed by the general definition of the scheduler, since the latter only checks for consistency, and not for coherence.

interpreter see [27]. We start this section by presenting the notation that is used in the specification of the interpreter.

3.1. 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 5 for reference);
   - *class* : NODE → CLASS returns the syntactical class of a node (i.e., the name of the corresponding grammar non-terminal class); for example RuleDecl
   - *token* : NODE → TOKEN returns the syntactical token represented by the node (i.e., either a keyword, an identifier, or a literal value); for example 123
   - *pattern* : NODE → PATTERN returns the symbolic name for the specific grammar pattern corresponding to the node; for example IfThen for the pattern **if** ... **then** ...
   - $[\![\cdot]\!]$: NODE → LOC × UPDATES × ELEMENT holds the result of the interpretation of a node, given by a triple formed by a location (that is, the l-value of an expression, when it is defined), a multiset of update instructions, and a value (that is, the r-value of an expression)[10]. 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. $\mathit{loc}(n) \equiv [\![n]\!] \downarrow 1$.
     - *updates* : NODE → UPDATES returns the updates associated to the given node, i.e. $\mathit{updates}(n) \equiv [\![n]\!] \downarrow 2$.
     - *value* : NODE → ELEMENT returns the value (r-value) associated to the given node, i.e. $\mathit{value}(n) \equiv [\![n]\!] \downarrow 3$.
     - *evaluated* : NODE → BOOLEAN indicates if a node has been fully evaluated. We have,

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

   - *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;

[10] 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.

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

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

     are to be intended as

     if *conditions* **then** *actions*

     where the *conditions* are derived from the pattern according to the conventions above, as more formally specified in Table 1; in the action part of such a rule, an unquoted and unbound occurrence of l is to be interpreted as the *loc* of the corresponding node; an unquoted and unbound occurrence of v is to be interpreted as the *value* of the corresponding node; an unquoted and unbound occurrence of u as the *updates* of the corresponding node; and an unquoted and unbound occurrence of *x* as the *token* of the corresponding node.

   Table 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

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

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

Table 1. Abbreviations in syntactic pattern-matching rules.

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

Table 2. Examples of how pattern matching notation is translated into ASM rules.

location, updates or value and assign it to $[\![pos]\!]$, thus implicitly returning control back to our parent. As exemplified in Table 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.

3.2. Kernel Interpreter

The kernel behavior of the interpreter which specifies the core constructs of the language is defined by KernelInterpreter rule (see ExecuteTree in Section 2.2). In this section, we present a definition of this rule in form of parallel composition of pattern-action rules, which ultimately defines the core syntax and semantics of the language. The definition is presented in two parts: patterns for expressions and patterns for rule forms.

3.2.1. 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 result in values, namely for evaluating literals (true, false, undef) and nullary or n-ary functions.

Literals are simply lifted to their semantic counterparts:

Kernel Expressions: Literals

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

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

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

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$(x, \langle\rangle)$

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

if *isFunctionName*(x) **then**
  choose $i \in [1..n]$ **with** $\neg$*evaluated*(λ_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 a 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:

HandleUndefinedIdentifier

HandleUndefinedIdentifier$(x, args) \equiv$
  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 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*$(l))$

3.2.2. Kernel Rule Interpreter

Rule plug-ins provide the semantics for executing 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 to 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 [16] 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[11] with the one for functional application) to handle rule calls.

Kernel Rules

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

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

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

[11] Mutual exclusion is due to the two guards *isFunctionName*(x) and *isRuleName*(x) which prevent execution of both rules on the same node.

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 [16] Section 4.1.2. Here, we model macro calls, while the effect of submachine calls can simply be achieved by using the **iterate** construct (see Section 3.4.1).

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

The rule CopyTreeSub returns a copy of the given parse tree, where every instance of an identifier node in a given sequence (formal parameters) is substituted by a copy of the corresponding parse tree in another sequence (actual parameters). We assume that the elements in the formal parameters list are all distinct (i.e., it is not possible to specify the same name for two different parameters). Also, formal parameters substitution is applied only to occurrences of formal parameters in the original tree passed as argument, and *not* also on the actual parameters themselves. A full definition of CopyTreeSub is provided in [27].

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

Kernel Rules: Assignment

$(\!|\ {}^{\alpha}\boxed{?} := {}^{\beta}\boxed{?}\ |\!)$ → **choose** $\tau \in \{\alpha, \beta\}$ **with** $\neg$*evaluated*(τ)
  $pos := \tau$
ifnone
  if *loc*$(\alpha) \neq$ *undef*
    $[\![pos]\!] := (undef, \{\!|\langle loc(\alpha), value(\beta)\rangle|\!\}, undef)$
  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** ${}^{\alpha}x$ **do** ${}^{\beta}\boxed{r}\ |\!)$ → **let** $e =$ *new*(ELEMENT) **in**
  env$(x) := e$
  $pos := \beta$

$(\!|$ **import** ${}^{\alpha}x$ **do** ${}^{\beta}u\ |\!)$ → *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 $\boxed{r}$ 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.

3.3. Standard Library

The CoreASM language is accompanied by a set of plug-ins which provide all the rule forms defined for Basic ASMs in [16] and several commonly used backgrounds with corresponding operations. These plug-ins form the *standard library* of CoreASM, which is implicitly loaded with any specification. Space limitations prevent us from providing a full account of the standard library here, but we present in this section a selection of some of these plug-ins to give an intuition of how plug-ins are specified.

3.3.1. Standard Rules

We initiate by presenting rule plug-ins for some of the rule forms defined for Basic ASMs. As the reader will recall, only assignment and **import** are defined in the kernel, with almost everything else being provided by these plug-ins. The most fundamental rule is the block-rule, specified as follows:[12]

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

BlockRule

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

Here, all the rules in a block are evaluated in an unspecified order, with the final result being the union of all the updates[13] produced by the various rules in the block.

Close in importance to the block-rule comes the **if**-rule. We accept a slightly extended syntax, where the guard is not restricted to be a formula (as per Definition 2.4.14 in [16]), but rather any expression that returns a Boolean. This guarantees that plug-ins will be able to extend the set of allowable guards as needed. Notice that this approach is conservative w.r.t. the standard definition, given that formulae in the sense of [16] are indeed expressions (supported by the Boolean and Quantifiers plug-ins in our standard library).

IfRule

$$\begin{array}{lcl}
(\!| \textbf{if } {}^{\alpha}\boxed{e} \textbf{ then } {}^{\beta}\boxed{r} |\!) & \rightarrow & \mathit{pos} := \alpha \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} |\!) & \rightarrow & \textbf{if } v = \mathsf{tt} \textbf{ then } \mathit{pos} := \beta \\
 & & \textbf{else if } v = \mathsf{ff} \textbf{ then } [\![\mathit{pos}]\!] := (\mathit{undef}, \{\!\}, \mathit{undef}) \\
 & & \textbf{else } \mathsf{Error}(\text{'Condition must be either true or false.'}) \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}u |\!) & \rightarrow & [\![\mathit{pos}]\!] := (\mathit{undef}, u, \mathit{undef}) \\
 & & \\
(\!| \textbf{if } {}^{\alpha}\boxed{e} \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}\boxed{r} |\!) & \rightarrow & \mathit{pos} := \alpha \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}\boxed{r} |\!) & \rightarrow & \textbf{if } v = \mathsf{tt} \textbf{ then } \mathit{pos} := \beta \\
 & & \textbf{else if } v = \mathsf{ff} \textbf{ then } \mathit{pos} := \gamma \\
 & & \textbf{else } \mathsf{Error}(\text{'Condition must be either true or false.'}) \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}u \textbf{ else } {}^{\gamma}\boxed{r} |\!) & \rightarrow & [\![\mathit{pos}]\!] := (\mathit{undef}, u, \mathit{undef}) \\
(\!| \textbf{if } {}^{\alpha}v \textbf{ then } {}^{\beta}\boxed{r} \textbf{ else } {}^{\gamma}u |\!) & \rightarrow & [\![\mathit{pos}]\!] := (\mathit{undef}, u, \mathit{undef})
\end{array}$$

To show how the local environment is modified, we present the specification of the **let**-rule:

LetRule

$$\begin{array}{lcl}
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}\boxed{e} \textbf{ in } {}^{\gamma}\boxed{r} |\!) & \rightarrow & \mathit{pos} := \beta \\
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}v \textbf{ in } {}^{\gamma}\boxed{r} |\!) & \rightarrow & \mathit{pos} := \gamma \\
 & & \mathit{env}(x) := v \\
(\!| \textbf{let } {}^{\alpha}x = {}^{\beta}v \textbf{ in } {}^{\gamma}u |\!) & \rightarrow & \mathit{env}(x) := \mathit{undef} \quad \text{// No nesting} \\
 & & [\![\mathit{pos}]\!] := (\mathit{undef}, u, \mathit{undef})
\end{array}$$

indentation to express nesting are common choices.

[13] More precisely, the multiset-union of the update instructions returned by the rules.

3.3.2. Standard Expressions

Commonly used expression forms are also added to the language by the standard library. This section exemplifies this approach by presenting three examples: the equivalence operator, the Boolean conjunction operator, and the integer *plus* operator.

In the equivalence operator, two values are considered to be equal if and only if at least one of their backgrounds regards them as equal. In the following rule, the equality function provided by the backgrounds of the operands is queried to determine whether they are equal.

Equivalence Operator

$\langle\!| {}^{\alpha}\boxed{?} = {}^{\beta}\boxed{?} |\!\rangle \rightarrow$
choose $\lambda \in \{\alpha, \beta\}$ **with** $\neg$*evaluated*(λ)
  pos $:= \lambda$
ifnone
  let $e_1 =$ *value*$(\alpha),\ e_2 =$ *value*(β) **in**
    let $b_1 =$ *bkg*$(e_1),\ b_2 =$ *bkg*(e_2) **in**
      if *equal*$_{b_1}(e_1, e_2) \vee$ *equal*$_{b_2}(e_2, e_1)$ **then**
        $[\![$*pos*$]\!] :=$ (*undef*, *undef*, tt)
      else
        $[\![$*pos*$]\!] :=$ (*undef*, *undef*, ff)

Boolean operators are also defined in the standard library. The following rule, as an example, defines the conjunction operator:

Boolean Operator: AND

$\langle\!| {}^{\alpha}\boxed{?} \wedge {}^{\beta}\boxed{?} |\!\rangle \rightarrow$
choose $\lambda \in \{\alpha, \beta\}$ **with** $\neg$*evaluated*(λ)
  pos $:= \lambda$
ifnone
  if (*value*$(\alpha) =$ tt$) \wedge$ (*value*$(\beta) =$ tt$)$ **then**
    $[\![$*pos*$]\!] :=$ (*undef*, *undef*, tt)
  else
    $[\![$*pos*$]\!] :=$ (*undef*, *undef*, ff)

In the same spirit, the following rule, provided by the Integer plug-in, defines the *plus* operator on integer values:

Integer Operator: Plus

$\langle\!| {}^{\alpha}\boxed{?} + {}^{\beta}\boxed{?} |\!\rangle \rightarrow$
choose $\lambda \in \{\alpha, \beta\}$ **with** $\neg$*evaluated*(λ)
  pos $:= \lambda$
ifnone
  let $v =$ *plus*$_{\text{int}}$(*value*(α), *value*(β)) **in**
    $[\![$*pos*$]\!] :=$ (*undef*, *undef*, v)

Other operators and standard backgrounds follow in the same vein, and we will not insist here on their formal definition.

CoreASM defines a special framework to handle operator extensions provided by plug-ins. Such a framework allows the engine to support operator overloading and reduces the redundancy of operand evaluation. Operator evaluation and extensions in CoreASM are presented in more detail in [45, Ch. 6].

3.4. Extensions

In addition to basic ASM rules, Turbo ASM rules presented in [16] are also defined in the CoreASM language. In this section we present rule plug-ins for sequentiality, iteration, and exception handling, following the syntax and semantics presented in [16]. Specifications have to explicitly import these plug-ins, differently from those in the standard library which are always implicitly imported. A number of other extension plug-ins for commonly used backgrounds and rule forms are distributed together with the CoreASM engine itself.

3.4.1. Sequentiality and Iteration Rules

Sequential execution of rules is modeled by the **seq**-rule. Since we want to model the effect of evaluating the second rule in a sequence in the state that would be produced by applying the updates produced by the first rule, we have to "simulate" the application of the updates, without really modifying the current state. This is obtained by using a *stack* of states, managed through three macros: PushState copies the current state in the stack, PopState retrieves the state from the top of the stack (thus discarding the current state), and Apply(u) applies the updates in the update set u to the current state. In addition, we will use the macro Diff to compute the update set representing the difference between the current state and the state at the top of the stack. Formal definitions for these macros are given in [27]. Based on the intuitive understanding of these macros, the interpreter plug-in for the **seq**-rule can be specified as follows:

SeqRule

$(\!|\, {}^{\alpha}\boxed{r}_1 \textbf{ seq } {}^{\beta}\boxed{r}_2 \,|\!) \quad \rightarrow \quad pos := \alpha$

$(\!|\, {}^{\alpha}u_1 \textbf{ seq } {}^{\beta}\boxed{r}_2 \,|\!) \quad \rightarrow$

```
if isConsistent(u1) then
  PushState
  Apply(u1)
  pos := β
else
  ⟦pos⟧ := (undef, u1, undef)
```

$(\!|\, {}^{\alpha}u_1 \textbf{ seq } {}^{\beta}u_2 \,|\!) \quad \rightarrow$

```
PopState
⟦pos⟧ := (undef, u1 ⊕ u2, undef)
```

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

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

The intuition behind the definition of $\oplus$ is that updates generated by the first rule are used, except in cases where the second rule provides a (subsequent) update to the same location.

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

given below:

Iterate Rule

$\langle\!|$ **iterate** $^{\alpha}\boxed{r}$ $|\!\rangle$ $\rightarrow$ PushState
pos $:= \alpha$

$\langle\!|$ **iterate** $^{\alpha}u$ $|\!\rangle$ $\rightarrow$ **if** $u = \{\!\}\ \vee\ \neg$*isConsistent*(u) **then**
$[\![$*pos*$]\!] := ($*undef*, Diff $\cup\ u$, *undef*$)$
PopState
else
Apply(u)
ClearTree(α)
pos $:= \alpha$

Notice here how iteration is carried on in a separate state, after saving the original one in the stack; after the iteration is completed, the difference between the initial and the final state is encoded as updates to the initial state, the initial state is restored from the stack, and the computed updates are returned. Notice also that, after each step in the iteration, the entire subtree is cleared (i.e., the $[\![\cdot]\!]$ function of each node is set to *undef*), so that the computation of the next step can proceed as usual.

3.4.2. Exception Handling

The **try/catch** construct introduced in [16] lets the specifier declare that inconsistent updates on certain locations should not abort the current run, but rather be "caught" and handled by executing a given rule. In particular, the informal semantics for the construct

$$\textbf{try}\ r\ \textbf{catch}\ l_1, \ldots, l_n\ \textbf{do}\ q$$

is to execute r, and if any inconsistent update is generated for any of the locations $l_1, \ldots, l_n$, the updates of r are discarded and q is executed instead. Formally, we specify the **try/catch** construct as follows:

ExceptionRule

$\langle\!|$ **try** $^{\alpha}\boxed{r}_1$ **catch** $^{\lambda_1}\boxed{l}_1, \ldots, ^{\lambda_n}\boxed{l}_n$ **do** $^{\beta}\boxed{r}_2$ $|\!\rangle\ \rightarrow$ *pos* $:= \alpha$

$\langle\!|$ **try** $^{\alpha}u_1$ **catch** $^{\lambda_1}\boxed{?}_1, \ldots, ^{\lambda_n}\boxed{?}_n$ **do** $^{\beta}\boxed{r}_2$ $|\!\rangle\ \rightarrow$
choose $i \in [1..n]$ **with** $\neg$*evaluated*(λ_i) **do**
pos $:= \lambda_i$
ifnone
if *isConsistent*$(u_1 \upharpoonright \{$*loc*$(\lambda_1), \ldots,$ *loc*$(\lambda_n)\})$
then
$[\![$*pos*$]\!] := ($*undef*$, u_1,$ *undef*$)$
else
pos $:= \beta$

$\langle\!|$ **try** $^{\alpha}u_1$ **catch** $^{\lambda_1}l_1, \ldots, ^{\lambda_n}l_n$ **do** $^{\beta}u_2$ $|\!\rangle\ \rightarrow\ [\![$*pos*$]\!] := ($*undef*$, u_2,$ *undef*$)$

where the $\restriction$ operator is defined as:

$$U \restriction H = \{u \in U \mid \mathit{location}(u) \in H\}$$

3.5. Custom Extensions

The CoreASM language can be extended with non-standard rule forms, adding new capabilities and improving the expressiveness of the language. In this section we present two such custom extensions, which are not part of the CoreASM library, adding respectively a parallel-case rule and a form of rules which return a value instead of an update set.

3.5.1. A Parallel-Case Rule

We present here the specification for a plug-in implementing a parallel form of **case** . The syntax is similar to the one that is used in [51], but the semantics is quite different. Instead of evaluating the first rule with a matching guard value, all the rules with matching guard values will be evaluated in parallel. In essence, this parallel-case rule acts as a block rule in which all child rules are guarded against a given value.

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

ParallelCaseRule

$(\!|\,\textbf{case}\ {}^{\alpha}\boxed{e}\ \textbf{of}\ \{{}^{\lambda_1}\boxed{e}_1 \to {}^{\lambda'_1}\boxed{r}_1; \ldots; {}^{\lambda_n}\boxed{e}_n \to {}^{\lambda'_n}\boxed{r}_n\}\,|\!) \ \to\ \ \mathit{pos} := \alpha$

$(\!|\,\textbf{case}\ {}^{\alpha}v\ \textbf{of}\ \{{}^{\lambda_1}\boxed{?}_1 \to {}^{\lambda'_1}\boxed{r}_1; \ldots; {}^{\lambda_n}\boxed{?}_n \to {}^{\lambda'_n}\boxed{r}_n\}\,|\!) \ \to$
$\quad$ **choose** i **in** $[1..n]$ **with** $\neg \mathit{evaluated}(\lambda_i)$
$\quad\quad \mathit{pos} := \lambda_i$

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

3.5.2. Rules with a return value

A frequent and idiomatic use of Turbo ASMs is to compute functions by executing a rule and then extracting a value from the resulting set of updates, rather than applying the updates to the state. The syntax provided in [16], however, is not particularly practical, as the computation is restricted to be a statement assigning a value to a given identifier, and so cannot be used inside a complex expression. For

example, one has to write

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

instead of the more natural

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

We propose here an alternative syntax and semantics, formally described by the following rules:

ReturnRule

$(\!\vert$ **return** $^{\alpha}\boxed{e}$ **in** $^{\beta}\boxed{r}$ $\vert\!)$	$\rightarrow$	*pos* $:= \beta$
$(\!\vert$ **return** $^{\alpha}\boxed{e}$ **in** $^{\beta}u$ $\vert\!)$	$\rightarrow$	PushState Apply(u) *pos* $:= \alpha$
$(\!\vert$ **return** $^{\alpha}v$ **in** $^{\beta}u$ $\vert\!)$	$\rightarrow$	PopState $[\![$*pos*$]\!] :=$ (*undef*, *undef*, v)

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

4. 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 [38, 42] and, even further back, to the Prolog-based interpreter by Angelica Kappel [44]. Other interpreters and compilers followed: the lean *EA* compiler [3] from Karlsruhe University, the *scheme*-interpreter [22] from Oslo University, and an experimental EA-to-C++ compiler developed at Paderborn University. Besides practical work on ASM tools, conceptual frameworks for more systematic implementations were developed. The work on the *evolving algebra abstract machine (EAM)* [19], 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) [46] and the *Xasm (Extensible ASM) language [1, 2] are both based on compilers, while the* ASM Workbench *[18] and* AsmGofer *[49] provide ASM interpreters.*[14] The

[14] We focus here on the more common and well-known ASM tools. For a complete overview, see also [16], Sect. 8.3.

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 fledged 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 towards the world of 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 [20, 17]. 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 [50], 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 [14].

Xasm is an open source project [2] 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 [15]. 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.

5. Conclusion

We have outlined in this paper the design of the CoreASM extensible execution engine for abstract state machines. The CoreASM engine 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.

The CoreASM project [31] is an Open Source project, and as such it is in continuous development. Currently, the execution engine can execute simple specifications, and various plug-ins for common backgrounds (e.g., numbers, sets, strings, booleans) are available. More specialized plug-ins have also been developed, including a plug-in adding support for real time and one for interfacing ASM specifications with Java class libraries (including the Java standard library with all its facilities). The CoreASM GUI is instead still in early development. We are considering re-designing the first GUI, which was implemented as a stand-alone application, and produce instead a plug-in for the Eclipse integrated development environment [24].

Acknowledgements

Our sincere appreciation to Egon Börger for many inspiring discussions and persistent encouragement on the CoreASM project, as well as his valuable feedback on an early draft version of this paper. We also thank Mashaal Memon for his contribution to and his active involvement in the development and implementation of the CoreASM Engine, and the anonymous reviewers for their precious improvement suggestions.

References

[1] Anlauff, M.: XASM – An Extensible, Component-Based Abstract State Machines Language, *Abstract State Machines: Theory and Applications* (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, Ed.), 1912, Springer-Verlag, 2000.

[2] Anlauff, M., Kutter, P.: *eXtensible Abstract State Machines*, XASM open source project: `http://www.xasm.org`.

[3] Beckert, B., Posegga, J.: leanEA: A Lean Evolving Algebra Compiler, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95) (H. K. Büning, Ed.), 1092, Springer, 1996.

[4] Beierle, C., Börger, E., Durdanovic, I., Glässer, U., Riccobene, E.: Refining Abstract Machine Specifications of the Steam Boiler Control to Well Documented Executable Code, in: *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control* (J.-R. Abrial, E. Börger, H. Langmaack, Eds.), number 1165 in LNCS, Springer, 1996, 62–78.

[5] Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms, *ACM Transactions on Computation Logic*, **4**(4), 2003, 578–651.

[6] Börger, E.: A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control, *CSL'89. 3rd Workshop on Computer Science Logic* (E. Börger, H. Kleine Büning, M. M. Richter, W. Schönfeld, Eds.), 440, Springer, 1990.

[7] Börger, E.: A Logical Operational Semantics of Full Prolog. Part II: Built-in Predicates for Database Manipulation, in: *Mathematical Foundations of Computer Science* (B. Rovan, Ed.), vol. 452 of *LNCS*, Springer, 1990, 1–14.

[8] Börger, E.: The Origins and the Development of the ASM Method for High Level System Design and Analysis, *Journal of Universal Computer Science*, **8**(1), 2002, 2–74.

[9] Börger, E.: The ASM Ground Model Method as a Foundation for Requirements Engineering, *Verification: Theory and Practice*, 2003.

[10] Börger, E.: The ASM Refinement Method, *Formal Aspects of Computing*, 2003, 237–257.

[11] Börger, E., Fruja, N. G., Gervasi, V., Stärk, R. F.: A high-level modular definition of the semantics of C#, *Theoretical Computer Science*, **336**(2/3), May 2005, 235–284.

[12] Börger, E., Glässer, U., Müller, W.: The Semantics of Behavioral VHDL'93 Descriptions, *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, IEEE CS Press, Los Alamitos, California, 1994.

[13] Börger, E., Glässer, U., Müller, W.: Formal Definition of an Abstract VHDL'93 Simulator by EA-Machines, in: *Formal Semantics for VHDL* (C. Delgado Kloos, P. T. Breuer, Eds.), Kluwer Academic Publishers, 1995, 107–139.

[14] Börger, E., Päppinghaus, P., Schmid, J.: Report on a Practical Application of ASMs in Software Design, *Abstract State Machines: Theory and Applications* (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, Ed.), 1912, Springer-Verlag, 2000.

[15] Börger, E., Riccobene, E., Schmid, J.: Capturing Requirements by Abstract State Machines: The Light Control Case Study, *Journal of Universal Computer Science*, **6**(7), 2000, 597–620.

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

[17] Castillo, G. D., Glässer, U.: Simulation and Validation of High-level Abstract State Machine Specifications, *Modelling and Simulation: A Tool for the Next Millenium – Proc. of the 13th European Simulation Multiconference* (H. Szczerbicka, Ed.), 2, June 1999.

[18] Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines, *Applied Formal Methods — FM-Trends 98* (D. Hutter, W. Stephan, P. Traverso, M. Ullmann, Eds.), 1641, Springer-Verlag, 1999.

[19] Del Castillo, G., Durdanović, I., Glässer, U.: An Evolving Algebra Abstract Machine, Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95) (H. K. Büning, Ed.), 1092, Springer, 1996.

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

[21] Del Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language, *Proceedings of the 6th International Conference TACAS 2000* (S. Graf, M. Schwartzbach, Eds.), 1785, Springer-Verlag, 2000.

[22] Diesen, D.: *Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages*, Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway, March 1995.

[23] E. Börger and W. Schulte: A Practical Method for Specification and Analysis of Exception Handling: A Java/JVM Case Study, *IEEE Transactions on Software Engineering*, **26**(10), October 2000, 872–887.

[24] Eclipse Foundation: Eclipse.org web site, Last visited May 2005, `http://www.eclipse.org/`.

[25] Eschbach, R., Glässer, U., Gotzhein, R., Prinz, A.: On the Formal Semantics of SDL-2000: A Compilation Approach Based on an Abstract SDL Machine, *Abstract State Machines: Theory and Applications* (Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, Ed.), 1912, Springer-Verlag, 2000.

[26] Farahbod, R., Gervasi, V., Glässer, U.: *Design and Specification of the CoreASM Execution Engine*, Technical Report SFU-CMPT-TR-2005-02, Simon Fraser University, February 2005.

[27] Farahbod, R., Gervasi, V., Glässer, U.: *Design and Specification of the CoreASM Execution Engine*, Technical report, Simon Fraser University, To be published in October 2005, Revised version of SFU-CMPT-TR-2005-02, February 2005.

[28] Farahbod, R., Glässer, U., Vajihollahi, M.: Specification and Validation of the Business Process Execution Language for Web Services, *Abstract State Machines 2004. Advances In Theory And Practice: 11th International Workshop (ASM 2004)* (W. Zimmermann, B. Thalheim, Eds.), Springer-Verlag, Germany, March 2004.

[29] Farahbod, R., Glässer, U., Vajihollahi, M.: *Abstract Operational Semantics of the Business Process Execution Language for Web Services*, Technical Report SFU-CMPT-TR-2005-04, Simon Fraser University, Feb. 2005, Revised version of SFU-CMPT-TR-2004-03, April 2004.

[30] Farahbod, R., Glässer, U., Vajihollahi, M.: A Formal Semantics for the Business Process Execution Language for Web Services, *Web Services and Model-Driven Enterprise Information Systems* (S. Bevinakoppa, et al., Eds.), INSTICC Press, Portugal, May 2005.

[31] Farahbod, R., et al.: *The CoreASM Project*, `http://www.coreasm.org`.

[32] Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to Generate Tests from ASM Specifications, *Abstract State Machines 2003*, Springer, 2003.

[33] Glässer, U., Gotzhein, R., Prinz, A.: The formal semantics of SDL-2000: status and perspectives, *Comput. Networks*, **42**(3), 2003, 343–358.

[34] Glässer, U., Gu, Q.-P.: Formal Description and Analysis of a Distributed Location Service for Mobile Ad Hoc Networks, *Theoretical Computer Science*, **336**, May 2005, 285–309.

[35] Glässer, U., Gurevich, Y., Veanes, M.: Abstract Communication Model for Distributed Systems, *IEEE Trans. on Soft. Eng.*, **30**(7), July 2004, 458–472.

[36] Gurevich, Y.: Evolving Algebras 1993: Lipari Guide, in: *Specification and Validation Methods* (E. Börger, Ed.), Oxford University Press, 1995, 9–36.

[37] Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms, *ACM Transactions on Computational Logic*, **1**(1), July 2000, 77–111.

[38] Gurevich, Y., Huggins, J.: Evolving Algebras and Partial Evaluation, *IFIP 13th World Computer Congress* (B. Pehrson, I. Simon, Eds.), I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994.

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

[40] Gurevich, Y., Tillmann, N.: Partial Updates, *Journal of Theoretical Computer Science*, **336**(2-3), 2005, 311–342.

[41] Huckle, T.: *Collection of Software Bugs*, Technical report, Technical University Munich, 2004, Last visited Sep. 2005, `http://www5.in.tum.de/~huckle/bugse.html`.

[42] Huggins, J.: *An Offline Partial Evaluator for Evolving Algebras*, Technical Report CSE-TR-229-95, University of Michigan, 1995.

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

[44] Kappel, A. M.: Executable Specifications Based on Dynamic Algebras, in: *Logic Programming and Automated Reasoning* (A. Voronkov, Ed.), vol. 698 of *Lecture Notes in Artificial Intelligence*, Springer, 1993, 229–240.

[45] Memon, M. A.: *Specification Language Design Concepts: Aggregation and Extensibility in CoreASM*, Master Thesis, Simon Fraser University, Burnaby, Canada, April 2006.

[46] Microsoft FSE Group: *The Abstract State Machine Language*, Last visited June 2003, `http://research.microsoft.com/fse/asml/`.

[47] Müller, W., Ruf, J., Rosenstiel, W.: An ASM Based SystemC Simulation Semantics., *SystemC - Methodologies and Applications* (W. Müller, J. Ruf, W. Rosenstiel, Eds.), Kluwer Academic Publishers, June 2003.

[48] R. Eschbach and U. Glässer and R. Gotzhein and M. von Löwis and A. Prinz: Formal Definition of SDL-2000: Compiling and Running SDL Specifications as ASM Models, *Journal of Universal Computer Science*, **7**(11), 2001, 1024–1049.

[49] Schmid, J.: *Executing ASM Specitications with AsmGofer*, Last visited Sep. 2005, `www.tydo.de/AsmGofer/`.

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

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