

Design and Specification of the CoreASM Execution Engine

Roozbeh Farahbod¹, Vincenzo Gervasi², and Uwe Glässer¹

¹ Computing Science, Simon Fraser University, Burnaby, B.C., Canada
{rfarahbo,glaesser}@cs.sfu.ca

² Dipartimento di Informatica, Università di Pisa, Italy
gervasi@di.unipi.it

SFU-CMPT-TR-2005-02
February 2005

Abstract

In this report we introduce a new research effort in making abstract state machines 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 ASM. We present 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.

Contents

1	Introduction	3
2	Overall Architecture	5
3	CoreASM Components	6
3.1	CoreASM modules	7
3.2	Plug-ins	8
4	An ASM specification for CoreASM	10
4.1	The execution cycle	10
4.2	Notation	14
4.3	Kernel rules	17
4.3.1	KernelExpressionInterpreter rules	17
4.3.2	KernelRuleInterpreter rules	18
4.4	Standard background plug-ins	19
4.4.1	A simple background: Booleans	19
4.4.2	The Finite sets background	19
4.5	Standard rule plug-ins	21
4.5.1	Basic rules	21
4.5.2	choose and forall rules	22
4.5.3	Sequentiality, macros and iteration rules	26
4.5.4	Submachine calls	30
4.5.5	Local state, return values, and exception handling	30
4.5.6	Syntactic shorthands	32
4.6	Extension rule plug-ins	34
4.6.1	Support for abstraction	34
4.6.2	Support for native method calls	35
4.7	State- and tree-manipulation macros	36
5	Conclusion	38

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 [5], the IEEE language VHDL [2], programming languages like JAVA [7] and C# [1], 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.

Model-based systems engineering naturally demands for abstract executable 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 [3], Section 8.3) that have been developed over the years. The most prominent one is AsmL (ASM Language)[6], 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-

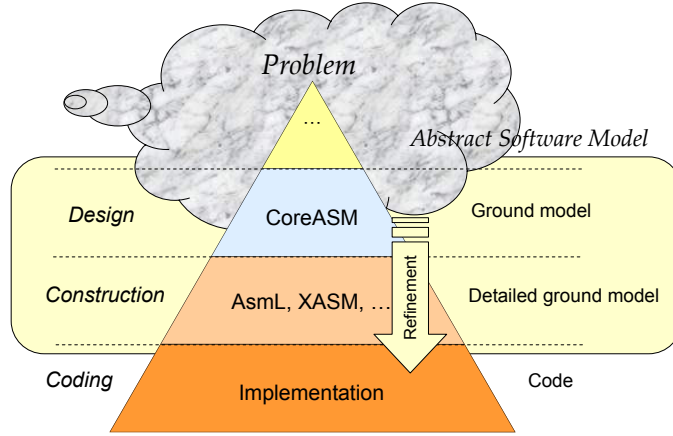


Figure 1: Background and Motivation

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 [3]. In this process, we aim at maintaining executability of even fairly abstract models. Another important characteristics 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 [4] 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 backgrounds¹, 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-ins* that allows to customize the language to specific needs.

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,

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

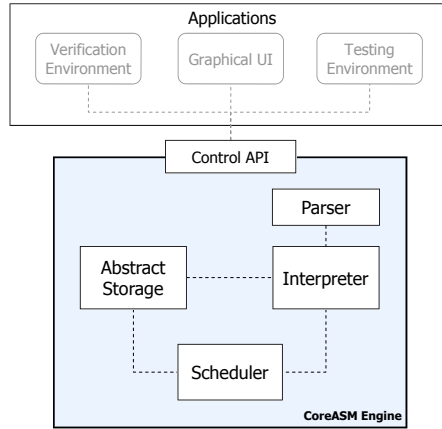


Figure 2: Overall Architecture of CoreASM

and for researchers that will be able to test in practice proposed extensions to the basic ASM language.

2 Overall Architecture

The CoreASM engine consists of four components: a *parser*, an *interpreter*, a *scheduler*, and an *abstract storage* (Figure 2). The interpreter, the scheduler, and the abstract storage work together to simulate an ASM run. The engine interacts with the environment through a single interface, called the *control API*, which provides various operations such as loading a CoreASM specification, starting an ASM run, or performing a single step.

The parser reads a CoreASM specification and 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 machine along with the history of its previous states. . 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 coordinates the execution of those agents in that step. 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 (Figure 5):

1. The Control API sends a STEP command to the scheduler.

2. The scheduler gets the whole set of agents from the abstract storage.
3. The scheduler selects a set of agents that will perform computation in the next step.
4. The scheduler selects a single agent and assigns it as the value of the *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. The interpreter notifies the scheduler that the interpretation is completed.
8. The scheduler then selects another agent in the selected set of agents. If there are no more agents left, it calls the abstract storage to fire the accumulated update set.
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.

3 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 3). 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 3.2, 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 dimension 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 spirit in which successive layers of the Java [7] and C# [1] languages have been used to structure the language definition into manageable parts.

²This 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.

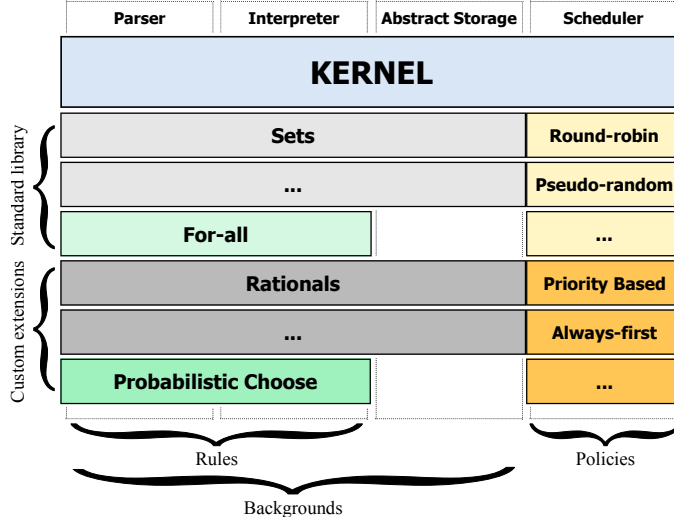


Figure 3: Layers and Modules of the CoreASM Engine

3.1 CoreASM modules

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 4, there are nodes that belong to the backgrounds of sets, integers, 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 interpreter executes programs and rules. It obtains an annotated parse tree from the parser and generates an update set. The interpreter interacts with the abstract storage to retrieve data from the current state and gradually creates the next update set. All the expressions are evaluated by the interpreter, possibly calling upon a background plug-in to perform the actual evaluation. Assignments are interpreted by evaluating the rhs of the assignment with respect to the current state, evaluating the location addressed by the lhs, and generating an update that will be returned as the result of the rule.

The abstract storage maintains a representation of the current state of the machine that is being simulated. It 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 sequential ASM, the scheduler merely arranges the execution of a step: it receives a *step* command from the control API, invokes the interpreter, and

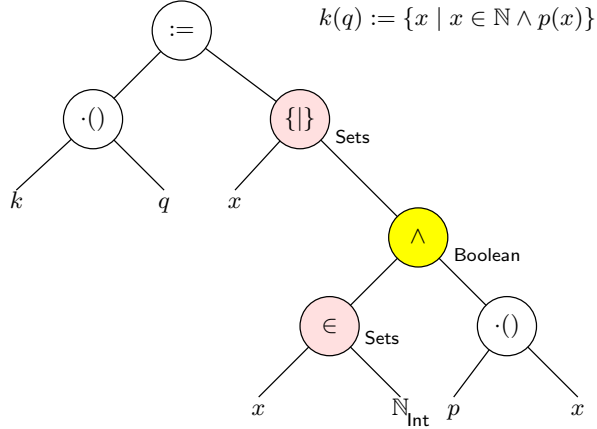


Figure 4: Sample Annotated Parse Tree

instruct the abstract storage to fire the 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, the scheduler also has to organize the execution of agents in each computation step. At the beginning of each DASM computation step, the scheduler chooses a subset of agents that will contribute to the computation of the next update set. The scheduler 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.

3.2 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 the basic 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 basic 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 execute³. 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.

Each class of plug-ins is characterized by an abstract interface, which is used by the CoreASM engine to communicate with the plugin. A simplified version of the various interfaces is shown in Section 4, whereas in the actual implementation a number of additional functions are needed for management purposes.

In CoreASM, the resident kernel (see Figure 3) only contains the bare essentials, that is, all that is needed to execute only the most basic ASM. As an ASM program is defined to be a finite set of rules, the two domains of *finite sets* and of *rules* are included in the kernel. Finite sets are represented through their characteristic functions, hence *functions* and *booleans* are 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 finite sets are in the kernel, infinite ones are implemented in a plug-in, which provides expression syntax for defining them (see the example in Figure 4), as well as an implicit representation for storing such 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: basic update instructions (i.e., assignments) and **import**. This particular choice is motivated by the fact that without updates 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.

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.

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 [3]), and by a set of specifications for writing new plug-ins that can easily be integrated in the environment. The latter must be explicitly imported into an ASM specification by an explicit directive, while the former are automatically imported in every specification by default.

³The policies in these plug-ins can also be called upon for implementing the **choose**-rule; to this end, we provide an extended version of **choose** that explicitly declares which policy to use.

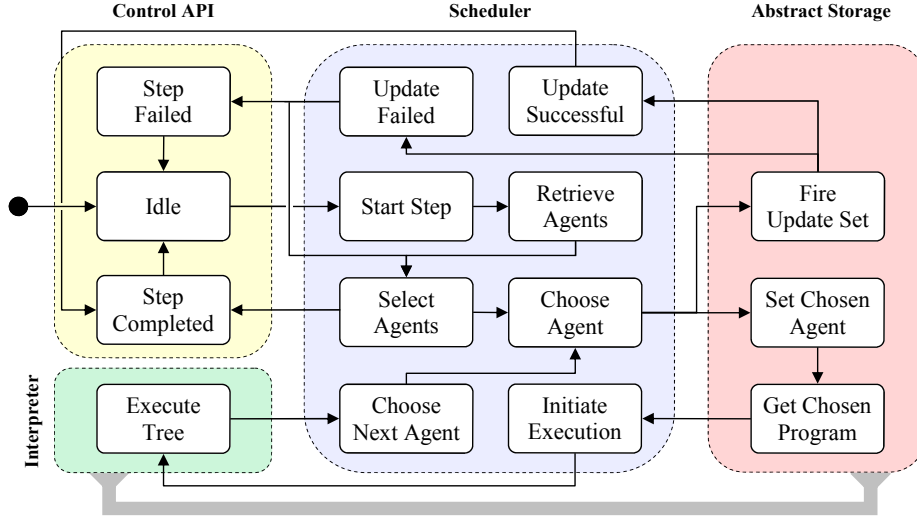


Figure 5: Lifecycle of a STEP command

4 An ASM specification for CoreASM

4.1 The execution cycle

In this section we present a high-level specification of how CoreASM performs one step of the simulated machine.⁴ The structure of the specification is that of a finite state automaton, as shown in Figure 5, whose current state is given by the variable *engineMode* (which is used in a **case** statement that controls which rules are executed). We present in the following the rules that are executed in each state (identifying state names with rule names).

The first state entered is the *Idle* state of the Control API:

Control API

Idle \equiv
if *stepCommand* **then**
 Next(*scStartStep*)

This rule simply waits for a “step” command from the environment (e.g., an interactive GUI or a debugger), to start the actual computation. We use the macro *Next* to transfer control to another state (values for *engineMode* are tagged with a 2-letter prefix indicating the module the state belongs to).

The *StartStep* rule in the scheduler simply initializes *updateSet* (the set of accumulated updates for the step) and *agentSet* (the current set of agents of the simulated machine). The latter is then assigned a value in the *RetrieveAgents* rule by querying the abstract storage module for the current value of *agents*

⁴The full specification, of course, models several other commands needed to implement a complete execution environment.

in the simulated machine. We model the query process through the abstract function $getValue(l)$ which takes a location and retrieves the value of the location from the simulated state. We use the notation $\langle\langle term \rangle\rangle$ to denote the quoted variable or literal term $term$ in the simulated machine.

The next rule, **SelectAgents**, chooses a set of agents to execute in the current step; if no agents are available, the step is considered complete. Otherwise, the **ChooseAgent** and **ChooseNextAgent** rules iterate over all selected agents. The former invokes, for each agent, the **SetChosenAgent** rule, that will ultimately come to the **ChooseNextAgent** rule. Computed updates are progressively added to $updateSet$, and when all agents have been run, control moves to **FireUpdateSet** in the abstract storage module.

Scheduler

```

StartStep  $\equiv$ 
   $updateSet := \{\}$ 
   $agentSet := undef$ 
  Next( $scRetrieveAgents$ )

RetrieveAgents  $\equiv$ 
   $agentSet := GetValue(\langle\langle agents \rangle\rangle, \langle \rangle)$ 
   $selectedAgentSet := undef$ 
  Next( $scSelectAgents$ )

SelectAgents  $\equiv$ 
  choose  $s$  with  $s \subseteq agentSet \wedge |s| \geq 1$  do
     $selectedAgentSet := s$ 
    Next( $scChooseAgent$ )
  ifnone
    Next( $caStepCompleted$ )

ChooseAgent  $\equiv$ 
  choose  $a$  in  $selectedAgentSet$  do
    remove  $a$  from  $selectedAgentSet$ 
     $chosenAgent := a$ 
    Next( $stSetChosenAgent$ )
  ifnone
    Next( $stFireUpdateSet$ )

ChooseNextAgent  $\equiv$ 
  add  $value(root(chosenProgram))$  to  $updateSet$ 
  Next( $scChooseAgent$ )

```

Two rules in the abstract storage module take care of setting the chosen agent (by assigning the value of $self$ in the simulated machine accordingly) 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 **InitiateExecution** rule.

```

SetChosenAgent  $\equiv$ 
  SetValue( $\langle\langle\text{self}\rangle\rangle, \langle\rangle$ ), chosenAgent)
  chosenProgram := undef
  Next(stGetChosenProgram)

```

```

GetChosenProgram  $\equiv$ 
  chosenProgram := GetValue( $\langle\langle\text{program}\rangle\rangle, \langle\langle\text{self}\rangle\rangle$ )
  Next(scInitiateExecution)

```

Following the footsteps of [3], we interpret a program by associating values (either elements of some domain or updates) and locations to nodes in the abstract syntax tree of the program. Before actually starting the interpreter, previously computed values are deleted by the `InitiateExecution` rule, and the initial position for the interpreter is set to the root node of the tree that represents the current program (that is, the program of the current agent, as established above).

```

InitiateExecution  $\equiv$ 
  pos := root(chosenProgram)
  forall n in nodes(chosenProgram) do
    value(n) := undef
    loc(n) := undef
  Next(inExecuteTree)

```

Due to space limitations, 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 in Section 3.2, nodes of the parse tree corresponding to grammar rules provided by a plug-in are annotated with the plug-in identifier (the annotation is modeled by the *plugin* function). 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 4.3) are used. As a result of the interpretation, *value(pos)* is set to either an abstract value (for expression nodes) or to a set of updates (for rule nodes).

```

ExecuteTree ≡
  if value(pos) = undef then
    if plugin(pos) ≠ undef then
      let R = pluginRule(plugin(pos)) in
        R
    else
      KernelRuleInterpreter
      KernelExpressionInterpreter
  else
    if parent(pos) = undef then
      Next(scChooseNextAgent)
    else
      pos := parent(pos)

```

After executing the programs of all the agents selected in the `SelectAgents` state, all the updates will have been accumulated in `updateSet`. Control will move from `ChooseAgent` to `FireUpdateSet` in the abstract storage module. The latter checks the consistency of the updates (possibly interacting with the relevant background plug-ins to evaluate equality), and either applies the updates to the current state, thus obtaining the next state, or provides an indication of failure.

```

FireUpdateSet ≡
  if consistent(updateSet) then
    ApplyUpdates
    Next(scSuccessfulUpdate)
  else
    Next(scUpdateFailed)

```

In that case (`UpdateFailed` rule), a different subset of agents can be tried. If all possible choices have been exhausted, the computation cannot proceed, and control moves to the `StepFailed` state in the control API. If instead updates can be applied successfully, the `StepCompleted` state of the control API is entered.

```

SuccessfulUpdate ≡
  Next(caStepCompleted)

UpdateFailed ≡
  if morePossibleSets then
    Next(scSelectAgents)
  else
    Next(scStepFailed)

```

In both cases, the following control API rules notify the environment of the success or failure of the step, and return to the `Idle` state awaiting for further

commands from the environment.

Control API

StepCompleted \equiv
NotifyEnvironment(*success*)
Next(*caIdle*)

StepFailed \equiv
NotifyEnvironment(*failure*)
Next(*caIdle*)

4.2 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 and updates. We state the following assumptions:

1. nodes in the tree are in the domain of the following (mostly partial) functions:
 - $first : Node \rightarrow Node$, $next : Node \rightarrow Node$, $parent : Node \rightarrow Node$ are static functions that implement tree navigation⁵; by using these functions, a rule can always access each of the children of a node, or go back to its parent;
 - $class : Node \rightarrow Class$ returns the syntactical class of a node (i.e., the name of the corresponding grammar rule);⁶
 - $token : Node \rightarrow Token$ returns the syntactical token represented by the node (e.g., either a keyword, an identifier, or a literal value);
 - $value : Node \rightarrow Val \cup Upd$ is the value (that is, the r-value) associated to an expression nodes, or a set of updates associated to a statement (rule) node;
 - $loc : Node \rightarrow Loc$ is the location (that is, the l-value) associated to nodes denoting functional applications;
 - $plugin : Node \rightarrow Bkg$ is the plug-in associated to expression and statement nodes, that is, the plug-in responsible for the 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 to concisely denote complex conditions on the nodes. In particular:

⁵We will silently skip over “empty” nodes, that derive from grammar productions only involving a single non-terminal, as in $Exp ::= Id$.

⁶

- we denote with e (expression) an unevaluated expression node; with v (value) an evaluated expression node (that is, an expression node whose *value* is not *undef*); with r (rule) an unevaluated statement node; and with u (uppdate set) an evaluated statement node. We will at times add subscripts to these variables, or use different names for special cases that will be discussed as appropriate;
- we denote with w a generic expression node (either evaluated or not), and with y a generic statement node (either evaluated or not);
- we denote with x an identifier node
- we denote with l (location) an evaluated expression for which a location has been computed (that is, whose *loc* is not *undef*);
- 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** αe **then** βr where α and β denote, respectively, the condition node and the then-part node of an if statement;
- rules of the form

$$(\textit{pattern}) \rightarrow \textit{actions}$$

are to be intended as

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

where the conditions are derived from the pattern according to the conventions above, and as fully specified in Table 1;

- an unquoted and unbound occurrence of v , u , w or y in the action part of such a rule is to be interpreted as the *value* of the corresponding node; an unquoted and unbound occurrence of x as the *token* of the corresponding node, and an unquoted and unbound occurrence of l as the *loc* 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 $env : Id \rightarrow Val$ global dynamic function
5. a static function $bkg : Val \rightarrow Bkg$ let us know, for any arbitrary value v , which background plug-in manages the value, or whether it is native in the core (if $bkg(v) = undef$).

Notice that, according to the rule `ExecuteTree` previously described, interpreter rules in the kernel or from plug-ins are only executed when $value(pos) = undef$, i.e. when the current node has not been fully evaluated yet. Control moves from node to node either by explicitly assigning to pos , or by setting $value(pos)$ to a value that is not *undef*; in that case, control is returned to the parent of pos (unless an explicit assignment to pos is also made at the same time).

Abbreviation	Condition (lhs)	Mention (rhs)
α_e	$class(\alpha) = \text{Exp} \wedge value(\alpha) = \text{undef}$	α
α_v	$class(\alpha) = \text{Exp} \wedge value(\alpha) \neq \text{undef}$	$value(\alpha)$
α_w	$class(\alpha) = \text{Exp}$	$value(\alpha)$
α_r	$class(\alpha) = \text{Rule} \wedge value(\alpha) = \text{undef}$	α
α_u	$class(\alpha) = \text{Rule} \wedge value(\alpha) \neq \text{undef}$	$value(\alpha)$
α_y	$class(\alpha) = \text{Rule}$	$value(\alpha)$
α_x	$class(\alpha) = \text{Id}$	$token(\alpha)$
α_l	$class(\alpha) = \text{Exp} \wedge loc(\alpha) \neq \text{undef}$	$value(\alpha)$
α, β etc.		$first(pos), next(first(pos)),$ etc.

Table 1: Abbreviations in syntactic pattern-matching rules.

Compact notation	Actual rule
$(\text{if } \alpha_e \text{ then } \beta_r) \rightarrow \dots$	if $class(pos) = \text{Rule}$ $\wedge token(pos) = \text{IfThen}$ $\wedge class(first(pos)) = \text{Exp}$ $\wedge value(first(pos)) = \text{undef}$ $\wedge class(next(first(pos))) = \text{Exp}$ $\wedge value(next(first(pos))) = \text{undef}$ then \dots
$(\text{if } \alpha_v \text{ then } \beta_r) \rightarrow \dots$	if $class(pos) = \text{Rule}$ $\wedge token(pos) = \text{IfThen}$ $\wedge class(first(pos)) = \text{Exp}$ $\wedge value(first(pos)) \neq \text{undef}$ $\wedge class(next(first(pos))) = \text{Rule}$ $\wedge value(next(first(pos))) = \text{undef}$ then \dots
$(\text{if } \alpha_v \text{ then } \beta_u) \rightarrow \dots$	if $class(pos) = \text{Rule}$ $\wedge token(pos) = \text{IfThen}$ $\wedge class(first(pos)) = \text{Exp}$ $\wedge value(first(pos)) \neq \text{undef}$ $\wedge class(next(first(pos))) = \text{Rule}$ $\wedge value(next(first(pos))) \neq \text{undef}$ then \dots
$(\alpha_x := \beta_v) \rightarrow state(x) := v$	if $class(pos) = \text{Rule}$ $\wedge token(pos) = :=$ $\wedge class(first(pos)) = \text{Id}$ $\wedge class(next(first(pos))) = \text{Exp}$ $\wedge value(next(first(pos))) \neq \text{undef}$ then $state(token(first(pos))) := value(next(first(pos)))$

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

4.3 Kernel rules

4.3.1 KernelExpressionInterpreter rules

As previously described, kernel rules implement the Boolean domain, assignments and import statements. Literals (true, false, undef) are lifted to their semantic counterparts:

	→		KernelExpressionInterpreter
$\langle \mathbf{true} \rangle$	→	$value(pos) := \mathbf{tt}$	
$\langle \mathbf{false} \rangle$	→	$value(pos) := \mathbf{ff}$	
$\langle \mathbf{undef} \rangle$	→	$value(pos) := \mathbf{uu}$	

The value of a simple variable is first looked into the set of in-scope local variables, and if none can be found there, the abstract storage is queried for a value (the result could be *undef*, that is lifted by the abstract storage to *uu*). Function references are handled in a similar way, except that they cannot be local variables. For elements of the state, the location is also stored in the tree, for later possible use by the assignment rule. Here, the abstract function *getValue(l)* (defined in Section 4.7) returns the value of location *l* in the current state of the simulated machine.

	→		KernelExpressionInterpreter
$\langle {}^\alpha x \rangle$	→	if $env(x) \neq \mathbf{undef}$ $value(pos) = env(x)$ else let $l = (x, \langle \rangle)$ in $value(pos) := \mathbf{GetValue}(l)$ $loc(pos) := l$	
$\langle {}^\alpha x(\lambda_1 w_1, \dots, \lambda_n w_n) \rangle$	→	choose $i \in [1..n]$ with $value(\lambda_i) = \mathbf{undef}$ $pos := \lambda_i$ ifnone let $l = (x, \langle w_1, \dots, w_n \rangle)$ in $value(pos) := \mathbf{GetValue}(l)$ $loc(pos) := l$	

Notice that in the latter rule the arguments w_i to the function f could be either expressions or values. The rule specifies that all arguments must be evaluated, without any specific order, to determine the location. While there are still unevaluated arguments, the rule would set *pos* to the node representing the unevaluated arguments; as soon as the evaluation of the argument is complete, control return 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 the *loc* and *value* for the node.

Also notice that functions for which an interpretation is defined in a background are not considered in the rules above, as the corresponding node will have its *plugin* attribute set by the parser to the plug-in implementing the given background, that will provide specific rules for computing the value (as shown

in the rules of page 13). Plug-in provided rules for interpreted constants and functions will typically follow the schema

$\langle \alpha x \rangle$	\rightarrow	$value(pos) := //$ interpretation of x
$\langle \alpha x(\lambda_1 w_1, \dots, \lambda_n w_n) \rangle$	\rightarrow	choose $i \in [1..n]$ with $value(\lambda_i) = undef$ $pos := \lambda_i$ ifnone $value(pos) := //$ interpretation of $x(w_1, \dots, w_n)$

4.3.2 KernelRuleInterpreter rules

The kernel of the CoreASM engine only includes two rule forms: assignment and **import**. Assignment is performed as follows:

$\langle \alpha w := \beta w \rangle$	\rightarrow	choose $\tau \in \{\alpha, \beta\}$ with $value(\tau) = undef$ $pos := \tau$ ifnone if $loc(\alpha) \neq undef$ $value(pos) := (loc(\alpha), value(\beta))$ else Error('Assignment to non-loc')	KernelRuleInterpreter
---------------------------------------	---------------	--	-----------------------

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:

$\langle \mathbf{import} \alpha x \mathbf{do} \beta r \rangle$	\rightarrow	$env(x) := \mathbf{NewElement}$ $pos := \beta$	KernelRuleInterpreter
$\langle \mathbf{import} \alpha x \mathbf{do} \beta u \rangle$	\rightarrow	$env(x) := undef$ $value(pos) := u$	

The value of the imported element x does not have a background (or it is a general background). The rationale is one cannot really **import** new elements with a pre-defined structure (such as String, Integer, or a TCP/IP Message). Either they are all in there from the beginning (e.g., Integers) or they need a factory (a constructor) to construct them (to form the required structure).

In the above rule, the **NewElement** macro returns a unique value for the imported element.

4.4 Standard background plug-ins

In this section we present the specification of a few background and rule plug-ins that are part of the standard library of CoreASM. We recall that each background plug-in provides an extension to the parser, an extension to the interpreter, and an extension to the abstract storage; rule plug-ins do not provide the latter. Here we present only the extensions to the interpreter; the corresponding extensions to the parser can also be easily inferred by observing the patterns used in the various rules.

4.4.1 A simple background: Booleans

The Boolean background provides operators implementing boolean algebra. Since the corresponding domain $\{\mathbf{true}, \mathbf{false}\}$ is already provided by the kernel, this plug-in turns out to be remarkably simple.

	BooleanBackground
$\langle \langle unop^\alpha e \rangle \rangle$	$\rightarrow pos := \alpha$
$\langle \langle unop^\alpha v \rangle \rangle$	$\rightarrow value(pos) := unop_{\mathbb{B}}(v)$
$\langle \langle {}^\alpha w_1 binop^\beta w_2 \rangle \rangle$	\rightarrow choose $\lambda \in \{\alpha, \beta\}$ with $value(\lambda) = undef$ $pos := \lambda$ ifnone $value(pos) := binop_{\mathbb{B}}(w_1, w_2)$

The $unop_{\mathbb{B}}$ and $binop_{\mathbb{B}}$ functions mentioned in the rules represent internal function of the Boolean plug-in that perform the corresponding semantic operation (e.g., **and**, **or**, **not**).

4.4.2 The Finite sets background

The background for finite sets represent sets by their characteristic functions, and provides a single literal (the empty set), constructors for sets, emptiness test and some set-theoretic operations. Although not shown here, the plug-in also implements the various enumeration functions that will be shown later and that are needed to support **choose**- and **forall**-rules.

	FiniteSetBackground
$\langle \langle \emptyset \rangle \rangle$	$\rightarrow value(pos) := \mathbf{EmptySet}$
$\langle \langle \{\lambda^1 w_1, \dots, \lambda^n w_n\} \rangle \rangle$	\rightarrow choose $i \in [1..n]$ with $value(\lambda_i) = undef$ $pos := \lambda_i$ ifnone import s do forall $i \in [1..n]$ do $member(s)(w_i) = true$ $value(pos) := s$

Testing for emptiness or membership is trivial:

	FiniteSetBackground
$\langle \mathbf{empty}^{\alpha e} \rangle \rightarrow$ $\langle \mathbf{empty}^{\alpha v} \rangle \rightarrow$	$pos := \alpha$ if $\text{dom}(\text{member}(v)) = \{\}$ then $value(pos) := \mathbf{tt}$ else $value(pos) := \mathbf{ff}$
$\langle {}^{\alpha}w_1 \in {}^{\beta}w_2 \rangle \rightarrow$	choose $\lambda \in \{\alpha, \beta\}$ with $value(\lambda) = \mathbf{undef}$ $pos := \lambda$ ifnone if $\text{member}(w_2)(w_1)$ then $value(pos) := \mathbf{tt}$ else $value(pos) := \mathbf{ff}$

Union, intersection, set difference are implemented as expected:

	FiniteSetBackground
$\langle {}^{\alpha}w_1 \cup {}^{\beta}w_2 \rangle \rightarrow$	choose $\lambda \in \{\alpha, \beta\}$ with $value(\lambda) = \mathbf{undef}$ $pos := \lambda$ ifnone import s from <i>Set</i> forall $x \in \text{dom}(\text{member}(w_1))$ do $\text{member}(s)(x) := \mathbf{true}$ forall $x \in \text{dom}(\text{member}(w_2))$ do $\text{member}(s)(x) := \mathbf{true}$ $value(pos) := s$
$\langle {}^{\alpha}w_1 \cap {}^{\beta}w_2 \rangle \rightarrow$	choose $\lambda \in \{\alpha, \beta\}$ with $value(\lambda) = \mathbf{undef}$ $pos := \lambda$ ifnone import s from <i>Set</i> forall $x \in \text{dom}(\text{member}(w_1))$ do if $\text{member}(w_2)(x)$ then $\text{member}(s)(x) := \mathbf{true}$ $value(pos) := s$
$\langle {}^{\alpha}w_1 \setminus {}^{\beta}w_2 \rangle \rightarrow$	choose $\lambda \in \{\alpha, \beta\}$ with $value(\lambda) = \mathbf{undef}$ $pos := \lambda$ ifnone import s from <i>Set</i> forall $x \in \text{dom}(\text{member}(w_1))$ do if $\neg \text{member}(w_2)(x)$ then $\text{member}(s)(x) := \mathbf{true}$ $value(pos) := s$

Finally, equivalence and inclusion are implemented as follows:

FiniteSetBackground

$$\begin{aligned}
\langle \alpha w_1 \text{relop}_{sets}^\beta w_2 \rangle &\rightarrow \text{choose } \lambda \in \{\alpha, \beta\} \text{ with } value(\lambda) = undef \\
&\quad pos := \lambda \\
&\text{ifnone} \\
&\quad \text{if } \text{dom}(member(w_1)) \text{relop}_{sets} \\
&\quad \quad \text{dom}(member(w_2)) \text{ then} \\
&\quad \quad \quad value(pos) := \text{tt} \\
&\text{else} \\
&\quad value(pos) := \text{ff}
\end{aligned}$$

4.5 Standard rule plug-ins

With the uttermost disregard for clarity, we can state that rule plug-ins provide rules for executing rules; execution of such rules results in a set of updates being the value for the rule node of the parse tree.

4.5.1 Basic rules

We initiate by presenting rule plug-ins for all the rule forms defined for Basic ASMs; we will then introduce some plug-in for Turbo ASMs and finally some custom plug-in for non-standard rules.

The semantics of the **skip** rule is simply to produce the empty set of updates:

SkipRule

$$\langle \text{skip} \rangle \rightarrow value(pos) := \{\}$$

The most fundamental rule is the block-rule, specified as follows:⁷

BlockRule

$$\begin{aligned}
\langle \lambda_1 y_1; \dots; \lambda_n y_n \rangle &\rightarrow \text{choose } i \in [1..n] \text{ with } value(\lambda_i) = undef \\
&\quad pos := \lambda_i \\
&\text{ifnone} \\
&\quad value(pos) := \bigcup_{i \in [1..n]} value(\lambda_i)
\end{aligned}$$

Close in importance 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 [3]), but rather any expression that returns **true**. This guarantees that plug-ins will be able to extend the set of allowable guards if needed. Notice that this approach

⁷We 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 $R \text{ par } Q$, meaning that P and Q are to be executed in parallel. Finally, also note that we are disregarding here the scope constructors provided by the grammar — either relying on braces $\{ \}$ or on indentation to express nesting are common choices.

is conservative w.r.t. the standard definition, given that formulae in the sense of [3] are indeed expressions (supported by the Boolean and Quantifiers plug-ins in our standard library).

	IfRule
$(\text{if } ^\alpha e \text{ then } ^\beta r)$	$\rightarrow \text{pos} := \alpha$
$(\text{if } ^\alpha v \text{ then } ^\beta r)$	$\rightarrow \text{if } v = \text{tt} \text{ then } \text{pos} := \beta \text{ else } \text{value}(\text{pos}) := \{\}$
$(\text{if } ^\alpha v \text{ then } ^\beta u)$	$\rightarrow \text{value}(\text{pos}) := u$
<hr/>	
$(\text{if } ^\alpha e \text{ then } ^\beta r \text{ else } ^\gamma r)$	$\rightarrow \text{pos} := \alpha$
$(\text{if } ^\alpha v \text{ then } ^\beta r \text{ else } ^\gamma r)$	$\rightarrow \text{if } v = \text{tt} \text{ then } \text{pos} := \beta \text{ else } \text{pos} := \gamma$
$(\text{if } ^\alpha v \text{ then } ^\beta u \text{ else } ^\gamma r)$	$\rightarrow \text{value}(\text{pos}) := u$
$(\text{if } ^\alpha v \text{ then } ^\beta r \text{ else } ^\gamma u)$	$\rightarrow \text{value}(\text{pos}) := u$

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

	LetRule
$(\text{let } ^\alpha x = ^\beta e \text{ in } ^\gamma r)$	$\rightarrow \text{pos} := \beta$
$(\text{let } ^\alpha x = ^\beta v \text{ in } ^\gamma r)$	$\rightarrow \text{env}(x) := v$
	$\text{pos} := \gamma$
$(\text{let } ^\alpha x = ^\beta v \text{ in } ^\gamma u)$	$\rightarrow \text{env}(x) := \text{undef} \quad // \text{ No nesting}$
	$\text{value}(\text{pos}) := u$

4.5.2 choose and forall rules

The **choose**-rule combines modification to the local environment with a delegation to the background plug-in providing the set-like structure among whose elements we want to choose. We present here the simplest form of **choose**, with no additional condition on the chosen value.⁸

⁸This, however, can be implemented by choosing in a set that already includes the condition.

$\langle \text{choose } ^\alpha x \in ^\beta e \text{ do } ^\gamma r \rangle$	→	$pos := \beta$
$\langle \text{choose } ^\alpha x \in ^\beta v \text{ do } ^\gamma r \rangle$	→	if $canChoose_{bkg(v)}(v)$ then if $empty_{bkg(v)}(v)$ then $value(pos) := \{\}$ else let $t = choose_{bkg(v)}(v)$ in $env(x) := t$ $pos := \gamma$ else Error('Cannot choose')
$\langle \text{choose } ^\alpha x \in ^\beta v \text{ do } ^\gamma u \rangle$	→	$env(x) := undef$ $value(pos) := u$
$\langle \text{choose } ^\alpha x \in ^\beta e \text{ do } ^\gamma r \text{ ifnone } ^\delta r \rangle$	→	$pos := \beta$
$\langle \text{choose } ^\alpha x \in ^\beta v \text{ do } ^\gamma r \text{ ifnone } ^\delta r \rangle$	→	if $canChoose_{bkg(v)}(v)$ then if $empty_{bkg(v)}(v)$ then $pos := \delta$ else let $t = choose_{bkg(v)}(v)$ in $env(x) := t$ $pos := \gamma$ else Error('Cannot choose')
$\langle \text{choose } ^\alpha x \in ^\beta v \text{ do } ^\gamma u \text{ ifnone } ^\delta r \rangle$	→	$env(x) := undef$ $value(pos) := u$
$\langle \text{choose } ^\alpha x \in ^\beta v \text{ do } ^\gamma r \text{ ifnone } ^\delta u \rangle$	→	$value(pos) := u$

The slightly more complete form of **choose** with an additional predicate can be specified as follows:

$\langle \mathbf{choose}^{\alpha} x \in^{\beta} e_1 \mathbf{with}^{\gamma} e_2 \mathbf{do}^{\delta} r \rangle$	→	$pos := \beta$
$\langle \mathbf{choose}^{\alpha} x \in^{\beta} v_1 \mathbf{with}^{\gamma} e_2 \mathbf{do}^{\delta} r \rangle$	→	<pre> if $canChoose_{bkg(v)}(v)$ then if $empty_{bkg(v)}(v)$ then $value(pos) := \{\}$ else if $canChoosePred(v)$ let $t = choosePred_{bkg(v)}(v, \gamma)$ in if $t \neq undef$ then $env(x) := t$ $pos := \delta$ else $value(pos) := \{\}$ else Error('Cannot choose') </pre>
$\langle \mathbf{choose}^{\alpha} x \in^{\beta} v_1 \mathbf{with}^{\gamma} e_2 \mathbf{do}^{\delta} u \rangle$	→	<pre> $env(x) := undef$ $value(pos) := u$ </pre>

This version, however, only works under the assumption that the background plug-in providing v implements the $choosePred$ function — that is, it can recursively call back the interpreter to evaluate the predicate. A version that does not require this capability can be obtained by refining the rules above as follows:

```

(choose  $^{\alpha}x \in ^{\beta}e_1$  with  $^{\gamma}e_2$  do $^{\delta}r$ )   $\rightarrow$    $pos := \beta$ 
                                            $alreadyTried(\beta) := \{\}$ 

(choose  $^{\alpha}x \in ^{\beta}v_1$  with  $^{\gamma}e_2$  do $^{\delta}r$ )   $\rightarrow$ 
  if  $canChoose_{bkg(v)}(v)$  then
    if  $empty_{bkg(v)}(v)$  then
      *  $value(pos) := \{\}$ 
    else if  $canChoosePred(v)$ 
      let  $t = choosePred_{bkg(v)}(v, \gamma)$  in
        if  $t \neq undef$  then
           $env(x) := t$ 
           $pos := \delta$ 
        else
           $value(pos) := \{\}$ 
    else // The plug-in does not implement  $choosePred$ 
      let  $t = chooseNext_{bkg(v)}(v, alreadyTried(\beta))$  in
        if  $t \neq undef$  then
           $alreadyTried(\beta) := alreadyTried(\beta) \cup \{t\}$ 
           $env(x) := t$ 
           $pos := \gamma$ 
        else
      *  $value(pos) := \{\}$ 
    else
      Error('Cannot choose')

(choose  $^{\alpha}x \in ^{\beta}v_1$  with  $^{\gamma}v_2$  do $^{\delta}r$ )   $\rightarrow$   if  $v_2 = tt$  then
                                            $pos := \delta$ 
                                           else
                                           ClearTree( $\gamma$ )

(choose  $^{\alpha}x \in ^{\beta}v_1$  with  $^{\gamma}v_2$  do $^{\delta}u$ )   $\rightarrow$    $env(x) := undef$ 
                                            $value(pos) := u$ 

```

The macro $ClearTree(\alpha)$ simply assigns $undef$ to $value$ and loc of all nodes in the subtree rooted at α (including the root).

A version of the **choose**-rule supporting both the **with** and the **ifnone** clause can be easily obtained by substituting the lines marked with \star with statements moving the control to the **ifnone** rule, as already shown.

The **forall**-rule is also similar; we show here only the simplest version with no predicate, given that the refinement needed to implement the **with** clause is similar to the one we have already discussed for **choose**.

```

(forall  $\alpha x \in \beta e$  do  $\gamma r$ )   $\rightarrow$   collectedUpdates(pos) := {}
                                         lastEnumerated( $\beta$ ) := undef
                                         pos :=  $\beta$ 

(forall  $\alpha x \in \beta v$  do  $\gamma r$ )   $\rightarrow$ 
  if canEnumeratebkg(v)(v) then
    let t = enumerateNextbkg(v)(v, lastEnumerated( $\beta$ )) in
      if t  $\neq$  undef then
        env(x) := t
        lastEnumerated( $\beta$ ) := t
        pos :=  $\gamma$ 
      else
        value(pos) := collectedUpdates(pos)
    else
      Error('Cannot enumerate')

(forall  $\alpha x \in \beta v$  do  $\gamma u$ )   $\rightarrow$ 
  collectedUpdates(pos) := collectedUpdates(pos)  $\cup$  {u}
  ClearTree( $\gamma$ )

```

4.5.3 Sequentiality, macros 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 (hypothetical) 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 state. This is obtained by using a *stack* of states, controlled by three macros: **Push** copies the current state in the stack, **Pop** retrieves the state from the top of the stack (thus discarding the current state), and **Apply**(*u*) applies the updates in the update set *u* to the current state. Formal definitions for these macros are given in Section 4.7. Based on the intuitive understanding of these macros, the interpreter plug-in for the **seq**-rule can be specified as follows:

```

( $\alpha r_1$  seq  $\beta r_2$ )   $\rightarrow$   pos :=  $\alpha$ 
( $\alpha u_1$  seq  $\beta r_2$ )   $\rightarrow$   if consistent(u1) then
                           Push
                           Apply(u1)
                           pos :=  $\beta$ 
                           else
                             value(pos) := u1
( $\alpha u_1$  seq  $\beta u_2$ )   $\rightarrow$   Pop
                           value(pos) := u1  $\oplus$  u2

```

The declaration and call of single rules (macros) are handled as follows. A special constructor **rule** is used to quote a rule, obtaining a value (see Sec-

tion 4.5.6 for notationally more convenient ways of treating macros):

MacroRule

```

(rule  $\alpha r$ )  $\rightarrow$  import  $t$  do // we should put  $t$  in Rules
     $body(t) := \alpha$ 
     $params(t) := \langle \rangle$ 
     $value(pos) := t$ 

(rule  $(\lambda^1 x_1, \dots, \lambda^n x_n) \alpha r$ )  $\rightarrow$  import  $t$  do
     $body(t) := \alpha$ 
     $params(t) := \langle x_1, \dots, x_n \rangle$ 
     $value(pos) := t$ 

```

Values obtained with the **rule** constructor (that is actually a λ operator) can be assigned or passed around as any other value. On values of type rule, two operations can be performed: *calling* them as macros, or *running* them as submachines. 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 [3] Section 4.1.2. We will see first the specification for macro calls, while submachine calls will be treated in Section 4.5.4.

MacroRule

```

(call  $\alpha e$ )  $\rightarrow$   $pos := \alpha$ 
(call  $\alpha v$ )  $\rightarrow$  let  $b = body(v)$  in
    if  $b \neq undef$  then
        if  $parent(b) = undef$  then
             $parent(b) := pos$ 
            ClearTree( $b$ )
             $pos := b$ 
        else
             $value(pos) := value(b)$ 
             $parent(b) := undef$ 
    else
        Error('Attempted call of a non-rule')

```

The case of a rule with parameters is only slightly more complex. ASMs differ from many other languages in that *call-by-name* 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 need to make all the changes on 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 [3], Definition 2.4.18). For simplicity, we do not explicitly check for such conditions in our specification.

The following rules for the Macro plug-in describe how definitions and calls for rules with parameters are handled.

	MacroRule
$\langle \mathbf{call}^{\alpha} e(\lambda_1 e_1, \dots, \lambda_n e_n) \rangle$	\rightarrow $pos := \alpha$
$\langle \mathbf{call}^{\alpha} v(\lambda_1 e_1, \dots, \lambda_n e_n) \rangle$	\rightarrow let $b = \mathit{body}(v)$, $p = \mathit{param}(v)$, $\Lambda = \langle \lambda_1, \dots, \lambda_n \rangle$ in if $b \neq \mathit{undef}$ then if $\mathit{workCopy}(pos) = \mathit{undef}$ then let $b' = \mathit{CopyTreeSub}(b, p, \Lambda)$ in $\mathit{workCopy}(pos) := b'$ $\mathit{parent}(b') := pos$ $pos := b'$ else $\mathit{value}(pos) := \mathit{value}(\mathit{workCopy}(pos))$ $\mathit{workCopy}(pos) := \mathit{undef}$ else Error('Attempted call of a non-rule')

The definition for the CopyTreeSub macro can be found in Section 4.7.

With the MacroRules plug-in, we have completely described the standard library of the CoreASM engine for Basic ASMs. One of the main concepts in Turbo ASMs is the notion of sequential composition and of iteration. For sequential composition, the definition for **seq** provided by [3] as Definition 4.1.1 is actually identical to the one for Basic ASMs given by Table 2.2 (although the latter is presented in structured operational semantics style, whereas the former is given in denotational semantics style), hence our SeqRule plug-in already provides all that is needed.

Iteration, however, requires a special treatment. 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).

	IterateRule
$\langle \mathbf{iterate}^{\alpha r} \rangle$	\rightarrow Push $pos := \alpha$
$\langle \mathbf{iterate}^{\alpha u} \rangle$	\rightarrow if $u = \{\}$ \vee $\neg consistent(u)$ then $value(pos) := Diff \cup u$ Pop else Apply(u) ClearTree(α) $pos := \alpha$

The non-standard **while**-rule can also be defined in a similar way. The semantics of a rule **while** ($cond$) R is to iterate the execution of R until $cond$ becomes false, or R produces an empty or inconsistent update set. Thus, the following equivalence holds:

$$\mathbf{while} (cond) R = \mathbf{iterate} \mathbf{if} cond \mathbf{then} R$$

The corresponding rule (which, as a matter of convenience, is implemented in the same plug-in as **iterate**) is thus

	IterateRule
$\langle \mathbf{while}^{(\alpha e) \beta r} \rangle$	\rightarrow Push $pos := \alpha$
$\langle \mathbf{while}^{(\alpha v) \beta r} \rangle$	\rightarrow if $v = \mathbf{tt}$ then $pos := \beta$ else $value(pos) := Diff$ Pop
$\langle \mathbf{while}^{(\alpha v) \beta u} \rangle$	\rightarrow if $u = \{\}$ \vee $\neg consistent(u)$ then $value(pos) := Diff \cup u$ Pop else Apply(u) ClearTree(α) ClearTree(β) $pos := \alpha$

Notice that other choices for the semantics of **while** were also possible: for example, [3] in Example 4.1.4 presents a variant that does not terminate when the update set produced by the rule is empty (their Example 4.1.2 is instead consistent with our definition).

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

4.5.4 Submachine calls

The semantics of submachine calls differs from that of simple macros, in that execution of the rule is iterated until the computation of the submachine is complete, and only at that point the cumulative updates are collected and lifted to the calling machine — which, effectively, sees the whole computation as a single step. In this sense, the semantics of submachine calls is rather similar to the semantics of XASMs turbo calls. To highlight this different semantics, we use the keyword **run** instead of **call** for invoking sub-machines.

	SubmachineCall
$(\mathbf{run}^{\alpha} e)$	$\rightarrow pos := \alpha$
$(\mathbf{run}^{\alpha} v)$	\rightarrow <pre> let $b = \mathit{body}(v)$ in if $b \neq \mathit{undef}$ then if $\mathit{parent}(b) = \mathit{undef}$ then Push ClearTree(b) $\mathit{parent}(b) := pos$ $pos := b$ else if $\mathit{value}(b) = \{\}$ $\vee \neg \mathit{consistent}(\mathit{value}(b))$ then $\mathit{value}(pos) := \mathit{Diff}$ $\mathit{parent}(b) := \mathit{undef}$ Pop else Apply($\mathit{value}(b)$) ClearTree(b) $pos := b$ else Error(‘Attempted run of a non-rule’) </pre>

Notice that according to the above definition, when a machine terminates a computation as a consequence of having generated an inconsistent set of updates the *previous* consistent state is considered as the result of the computation. An alternative semantics could be to incorporate the inconsistent updates in the result set, thus implying that a submachine that terminates due to an inconsistent set of updates being generated will produce an inconsistent set of updates in the calling machine, too.

The rule for running a parametric submachine is a variation of the rule above, with the addition that actual parameters are substituted for formal parameters in the body of the machine, as we already did for macro calls. For the sake of brevity, we do not present here the entire rule.

4.5.5 Local state, return values, and exception handling

Local state is introduced in rules by a special syntax ([3] page 169), which introduces both function names for local state and their initialization by means of rules. Updates made to these special locations are then discarded before

returning the final update set to the caller. In the same spirit, return values are simulated by designating a special location in the state, and by using the last update to that location as return value.

We sketch here only the basic idea of how local state and return values are handled. In particular, we omit the details of how local state initialization is performed, based on the observation that a declaration of local state with initialization can be transformed into a declaration without initialization followed by an explicit assignment (see Section 4.5.6 for details). The following rules describe how local state is implemented:

	LocalRule
$\langle \mathbf{local}^{\lambda_1} f_1 \dots \lambda_n f_n \mathbf{in}^{\alpha_r} \rangle \rightarrow pos := \alpha$ $\langle \mathbf{local}^{\lambda_1} f_1 \dots \lambda_n f_n \mathbf{in}^{\alpha_u} \rangle \rightarrow value(pos) := u \ominus \{f_1, \dots, f_n\}$	

where the \ominus operator is defined as follows:

$$u \ominus h = \{((f, a), v) \in u \mid f \notin h\}$$

As for result values, the following rules apply:

	ReturnRule
$\langle \alpha_r \mathbf{return}^{\beta} e \rangle \rightarrow pos := \alpha$ $\langle \alpha_u \mathbf{return}^{\beta} e \rangle \rightarrow \begin{array}{l} \mathbf{Push} \\ \mathbf{Apply}(u) \\ pos := \beta \end{array}$ $\langle \alpha_u \mathbf{return}^{\beta} v \rangle \rightarrow \begin{array}{l} \mathbf{Pop} \\ value(pos) := v \end{array}$	

Notice that these rules differ from the semantics given in [3] in Definition 4.1.7, where the result must be necessarily a single location (we allow for arbitrary expressions instead). According to the rules sketched above, it would not be possible to use any location of the local state in the expression to return. In fact, if we parenthesize the expression **local** x **in** r **return** e as in **(local** x **in** r) **return** e then x would not be visible in e , whereas if we parenthesize it as **local** x **in** (r **return** e) then the part r **return** e would be an expression, not a rule, and hence our semantics for **local** does not apply.

This, however, does not constitute a problem in itself, since our semantics for **return** discards *all* the updates in the rule, and hence *all* the state changes performed in the rule part of a **return** rule can be considered local. In this sense, a **return** guarantees the absence of side effects. Overall, our semantics seems more compositional than the one proposed in [3], thus we allow for this minor departure. Notice that the **return** rule also provides for easy integration of derived functions, and that the **call** rule returns a value when applied to a macro whose body is a **return** rule. For example, we could define a derived function as follows:

$$plus := \mathbf{rule} (x, y) \mathbf{skip return} x + y$$

and invoke it with

$$five := plus(2, 3)$$

(the definition can be actually written as $plus(x, y) \equiv x + y$ using the simplified syntax we describe in Section 4.5.6).

Finally, we specify the **try/catch** construct as follows:

	ExceptionRule
$(\text{try } ^\alpha r_1 \text{ catch } ^{\lambda_1} x_1 \dots ^{\lambda_n} x_n \text{ do } ^\beta r_2)$	$\rightarrow pos := \alpha$
$(\text{try } ^\alpha u_1 \text{ catch } ^{\lambda_1} x_1 \dots ^{\lambda_n} x_n \text{ do } ^\beta r_2)$	\rightarrow if $consistent(u_1 \setminus (u_1 \ominus \{x_1, \dots, x_n\}))$ then $value(pos) := u_1$ else $pos := \beta$
$(\text{try } ^\alpha u_1 \text{ catch } ^{\lambda_1} x_1 \dots ^{\lambda_n} x_n \text{ do } ^\beta u_2)$	$\rightarrow value(pos) := u_2$

4.5.6 Syntactic shorthands

In keeping with our tenet that CoreASM should be a concise and expressive language, we define a number of special syntactic shorthands to simplify writing specifications.

For rule declarations, we treat the form

$$x(x_1, \dots, x_n) \equiv r$$

as

$$x := \mathbf{rule} (x_1, \dots, x_n) r$$

(and analogously for the parameterless version). Moreover, we provide for the explicit declaration of the intended use of a rule (either as macro or as a sub-machine) in the definition of the rule itself. The following notation is used:

	MacroShorthands
$(\mathbf{macro} \ ^\alpha e)$	$\rightarrow pos := \alpha$
$(\mathbf{macro} \ ^\alpha v)$	\rightarrow if $body(v) \neq undef$ then $isMacro(v) := true$ $value(pos) := v$ else Error (‘Non-rule cannot be a macro’)
$(\mathbf{machine} \ ^\alpha e)$	$\rightarrow pos := \alpha$
$(\mathbf{machine} \ ^\alpha v)$	\rightarrow if $body(v) \neq undef$ then $isMachine(v) := true$ $value(pos) := v$ else Error (‘Non-rule cannot be a machine’)

Declared as	Treated as		
	a simple value	a macro call	a machine call
$x := \mathbf{rule\ r}$	x	$\mathbf{call\ x}$	$\mathbf{run\ x}$
$x := \mathbf{macro\ rule\ r}$		x	$\mathbf{run\ x}$
$x := \mathbf{machine\ rule\ r}$		$\mathbf{call\ x}$	x

Table 3: Syntactic shorthands for macro and TurboASM declarations.

Hence, an unannotated occurrence of a value of type rule can be implicitly treated as a **call** or a **run** without need for an explicit keyword, according to standard ASM practice, as shown in Table 3.

This implicit treatment needs a refinement of the `KernelExpressionInterpreter` macros for simple identifiers (e.g., x) and for functions (e.g., $x(a)$), which now assume a different semantics if x refers to a macro or machine rule. For example, the rule for a parameter-less identifier becomes

	KernelExpressionInterpreter
$(\langle^{\alpha} x \rangle) \rightarrow$ <pre> if $env(x) \neq undef$ $value(pos) = env(x)$ else let $l = (x, \langle \rangle), v = GetValue(l)$ in $loc(pos) := l$ if $isMachine(v) = true$ then MachineCall(v) else if $isMacro(v) = true$ then MacroCall(v) else $value(pos) := GetValue(l)$ </pre>	

where the `MachineCall` and `MacroCall` macros consist essentially of the body of the corresponding rules for **call** and **run**. The rule for $x(e_1, \dots, e_n)$ is refined in an analogous manner.

A similar approach is used for defining derived functions: the syntactic form

$$x(x_1, \dots, x_n) \equiv e$$

is translated as

$$x := \mathbf{macro\ rule\ } (x_1, \dots, x_n) \mathbf{\ skip\ return\ } e$$

Regarding initialization of local state, we rely on the following syntactical transformation:

$$\mathbf{local\ } f_1[Init_1] \dots \mathbf{local\ } f_k[Init_k] \mathbf{\ } body \rightarrow \mathbf{local\ } f_1, \dots, f_k \mathbf{\ in\ } Init_1; \dots; Init_k \mathbf{\ seq\ } body$$

A similar problem arises for the definition of the initial state of the machine. We assume that initialization is done by explicit assignments, which are evaluated in an “empty” state before the *Main* rule for the machine is called. The

general top-level structure of a CoreASM specification is thus

```

let  $Init = rule_{init}, Main = rule_{main}, R_1 = rule_1, \dots, R_n = rule_n$ 
in  $Init$  seq  $Main$ 

```

Appropriate lexical means are provided to convert the notion of *module* into this form of rule, which is then executed as usual. In particular, keyword markers like **init** and **main** could be used to identify the initialization and main rules in a CoreASM specification, with the understanding that in case several rules are marked as **init** or **main**, these will be collected and executed in parallel at appropriate times. If no rule is marked **init**, then the initialization rule is **skip** (that is, no initialization is performed beyond what the various plug-ins provide as initial state). If no rule is marked as **main**, the the main rule is the parallel composition of all the rules in the specification.

To facilitate simulation and debugging, the Control API provides an interface through which external applications can select which rules are to be considered as initial or main, regardless of the way they have been declared.

In the same spirit, a form of declaration of agent with contextual initialization of its program, which is given the shortened syntactical form

```

agent  $name \equiv rule$ 

```

is converted into an explicit import and assignment rule of the form

```

init import  $agent$  do
  add  $agent$  to  $Agents$ 
   $program(agent) := rule$ 
   $name(agent) := name$ 

```

Notice that, since the rule above is marked **init**, all agents will be already present, with the corresponding names and programs, in the initial state of the machine.

4.6 Extension rule plug-ins

In this section we present an example of non-standard plug-ins, i.e. plug-ins that implement rules not introduced in [3].

4.6.1 Support for abstraction

CoreASM provides support for abstraction by allowing the specifier to omit the body of a rule (most typically, a macro). This is obtained by providing a special rule **abstract** that acts as a *marker* for the execution of a yet-to-be-defined macro. The **abstract** rule includes an expression (most typically, a string) that can be used to identify what the abstract rule is intended to do.

$\langle \mathbf{abstract}^{\alpha} e \rangle$	\rightarrow	$pos := \alpha$
$\langle \mathbf{abstract}^{\alpha} v \rangle$	\rightarrow	$\mathbf{let} \ l = (\mathit{abstractUpdates}, \langle v \rangle) \ \mathbf{in}$ $\quad \mathit{value}(pos) := \{(l, \mathbf{tt})\}$

The way abstract rules works is the following. An abstract macro or sub-machine is declared to have an abstract body, for example:

ABSTRACTMACRO := **rule** (x) **abstract** ‘Working on ’ + x

or, using the traditional syntax,

ABSTRACTMACRO(x) \equiv **abstract** ‘Working on ’ + x

When ABSTRACTMACRO is invoked, the **abstract** rule will be executed, producing an update to the designated shared function *abstractUpdates* — effectively adding the value of the expression provided in the abstract declaration to the set *abstractUpdates*. It is intended that, at each step, the environment will read (possibly notifying the user) and clear the *abstractUpdates* set, that thus will contain at each step, an informal abstract description of the updates that are intended to be performed by the as-yet abstract rule.

4.6.2 Support for native method calls

For practical purposes, it might be useful to be able to call “native” methods⁹ that provides an escape mechanism from the CoreASM world.

As CoreASM already provides extension mechanisms for expressions and rules (by writing appropriate plug-ins), we do not intend for native methods to either generate updates sets or values. Rather, native methods are simply actions to be performed *outside* the CoreASM environment, that in no way alter the abstract state or the computation flow. They, however, can take arguments, that for the sake of simplicity and without loosing generality we will consider to be all strings.

The following rules specify the way native methods are called:

$\langle \mathbf{native}^{\alpha} x_1.{}^{\beta} x_2(\lambda_1 e_1, \dots, \lambda_n e_n) \rangle$	\rightarrow	NativePlugin
		$\mathbf{choose} \ i \in [1..n] \ \mathbf{with} \ \mathit{value}(\lambda_i) = \mathit{undef}$
		$pos := \lambda_i$
		ifnone
		$\mathbf{let} \ \mathit{args} = \langle \mathit{sval}(\lambda_1), \dots, \mathit{sval}(\lambda_n) \rangle \ \mathbf{in}$
		$\mathbf{CallNative}(x_1, x_2, \mathit{args})$
		$\mathbf{where} \ \mathit{sval}(\lambda) \equiv \mathit{stringValue}(\mathit{value}(\lambda))$

In the rule above, the two identifiers x_1 and x_2 are respectively the fully qualified name of a Java class, and the name of a static method of the given class.

⁹From the point of view of CoreASM, native methods are methods of Java classes; these in turn can call *really* native methods, e.g. in C or C++, by using the JNI APIs provided by Java.

Hence, only static methods returning void can be called through the **native** interface. A typical example of such a method would be `System.out.println()`.

The syntax for invoking native methods is *intentionally* prominent, to ensure that calls to native methods stand out of the surrounding context. Their semantics is *intentionally* limited to pure actions, to make sure that a native call cannot alter the ASM semantics of a specification.

4.7 State- and tree-manipulation macros

We model the simulated abstract state as a function $abstractState : Loc \rightarrow Val$, where locations are defined, as usual, by pairs of function names and arguments. With this assumption, the macros for manipulating the state are as follows (we are assuming $asPtr = 0$ in the initial state):

```

GetValue( $l$ )  $\equiv$ 
  if  $abstractState(l) = undef$ 
    return  $uu$ 
  else
    return  $abstractState(l)$ 

SetValue( $l, v$ )  $\equiv$ 
   $abstractState(l) := v$ 

Apply( $u$ )  $\equiv$ 
  forall  $(l, v) \in u$  do
    SetValue( $l, v$ )

Push  $\equiv$ 
   $asStack(asPtr) := abstractState$ 
   $asPtr := asPtr + 1$ 

Pop  $\equiv$ 
   $abstractState := asStack(asPtr - 1)$ 
   $asPtr := asPtr - 1$ 

Diff  $\equiv$ 
  let  $s = abstractState, s' = asStack(asPtr - 1)$  in
  return  $s - s'$ 

```

Tree manipulations functions are used throughout the interpreter for various house-keeping purposes. Here they are defined as TurboASM recursive rules:

```

ClearTree( $\alpha$ )  $\equiv$ 
  if  $\alpha \neq \text{undef}$  then
     $\text{value}(\alpha) := \text{undef}$ 
     $\text{loc}(\alpha) := \text{undef}$ 
    ClearTree( $\text{first}(\alpha)$ )
    ClearTree( $\text{next}(\alpha)$ )

```

```

CopyTree( $\alpha$ )  $\equiv$ 
  if  $\alpha \neq \text{undef}$  then
    let  $n = \text{new}(\text{Node})$  in
       $\text{first}(n) := \text{CopyTree}(\text{first}(\alpha))$ 
       $\text{next}(n) := \text{CopyTree}(\text{next}(\alpha))$ 
       $\text{class}(n) := \text{class}(\alpha)$ 
       $\text{token}(n) := \text{token}(\alpha)$ 
       $\text{plugin}(n) := \text{plugin}(\alpha)$ 
      return  $n$ 
    else
      return  $\text{undef}$ 

```

A specialized version of `CopyTree` called `CopyTreeSub` returns a copy of the given parse tree, where every instance of an `ld` 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 to formal parameters in the original tree passed as argument, and *not* also on the actual parameters themselves.

```

CopyTreeSub( $\alpha, \langle x_1, \dots, x_n \rangle, \langle \lambda_1, \dots, \lambda_n \rangle$ )  $\equiv$ 
  if  $\alpha \neq \text{undef}$  then
    if  $\text{class}(\alpha) = \text{ld} \wedge \exists i \text{ s.t. } \text{token}(\alpha) = x_i$  then
      return CopyTree( $\lambda_i$ )
    else
      let  $n = \text{new}(\text{Node})$  in
         $\text{first}(n) := \text{CopyTreeSub}(\text{first}(\alpha), \langle x_1, \dots, x_n \rangle, \langle \lambda_1, \dots, \lambda_n \rangle)$ 
         $\text{next}(n) := \text{CopyTreeSub}(\text{next}(\alpha), \langle x_1, \dots, x_n \rangle, \langle \lambda_1, \dots, \lambda_n \rangle)$ 
         $\text{class}(n) := \text{class}(\alpha)$ 
         $\text{token}(n) := \text{token}(\alpha)$ 
         $\text{plugin}(n) := \text{plugin}(\alpha)$ 
        return  $n$ 
    else
      return  $\text{undef}$ 

```

5 Conclusion

We have outlined in this document 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.

Acknowledgment. The second author wishes to acknowledge the financial support of the School of Computing Science at SFU which sponsored the visit during which this work was conducted.

References

- [1] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004. (In press, available online 18 December 2004).
- [2] 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.
- [3] E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [4] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

- [5] ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition*. International Telecommunication Union, 2001.
- [6] Microsoft FSE Group. *The Abstract State Machine Language*. cited June 2003, <http://research.microsoft.com/fse/asml/>.
- [7] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.