CoreASM: An Extensible ASM Execution Engine

Roozbeh Farahbod[1], Vincenzo Gervasi[2], and Uwe Glässer[1]

[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

Abstract. In this paper 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. 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.

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 for bridging the gap between informal requirements and precise specifications in the earlier phases of system design. The same advantages also simplify the task of constructing models of requirements that are being extracted from implementations in reverse engineering. 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 [1]. Widely recognized applications include semantic foundations of industrial system design languages like the ITU-T standard for SDL [7], the IEEE language VHDL [3], programming languages like JAVA [9] and C# [2], 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.

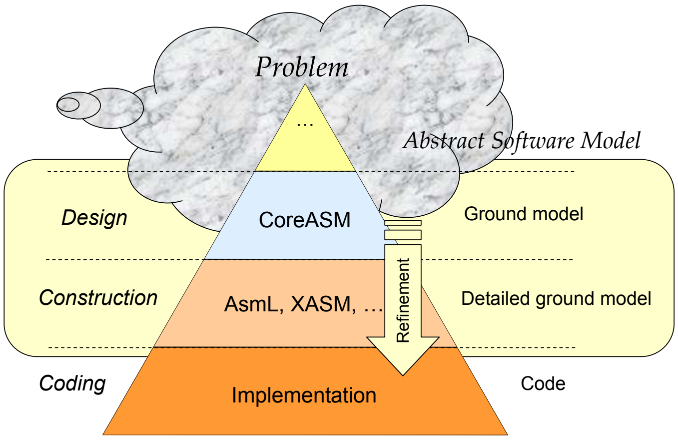

Fig. 1. Background and Motivation.

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 [4], Section 8.3) that have been developed over the years. The most prominent one is AsmL (ASM Language)[8], developed by the FSE group at Microsoft Research. AsmL is an executable language based on the concept of ASMs but also incorporates numerous object oriented features, thus departing in this respect from the theoretical model of ASMs, and comes with the richness of a fully fletched programming language. It also lacks any built-in support for dealing with distributed systems. Its design was shaped by the practical needs of dealing with fairly complex requirements and design specifications for the purpose of software testing; it can be thus said that its primary concerns are toward the world of code. This has made it less suitable for initial modeling at the peak of the problem space and also reduces the freedom of experimentation.

The CoreASM language and tool architecture focus on early phases of the software design process, and CoreASM primary concerns are toward the world of problems. In particular, we want to encourage rapid prototyping with ASMs, starting with mathematically-oriented, abstract and untyped models and gradually refining them down to more concrete versions — a powerful specification technique that has been exploited in [4]. 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 [6] 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[3], often with non-standard operations. We want to preserve in our language the freedom of experimentation that has proved so fruitful in the development of ASM concepts, and to this end we designed our architecture around the concept of *plug-in*s that allows to customize the language to specific needs.

An extensible, platform independent tool package (the language, its engine, 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 a high-level overview of the architecture of the CoreASM engine; details of its components are presented in Section 3, together with a discussion of the extensibility provisions in the architecture. Section 4 gives an abstract specification of how the CoreASM engine performs one step of the simulated machine, and Section 5 concludes the paper.

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: load a specification, start a run, perform a single step, get and set the value of a location, access the update set produced by a step, et cetera. The operations offered by the control API provide the foundation for integrating external tools with the execution engine, thus realizing an open tool architecture for ASMs. We are currently working on the design and implementation of a graphical execution monitor and debugger. Other useful applications include experimental validation by means of scripted execution (test suites) and simulation.

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

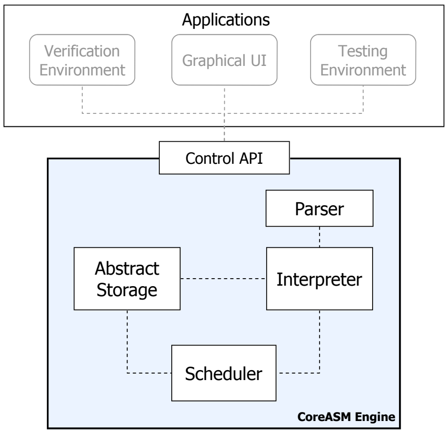

Fig. 2. Overall Architecture of the CoreASM engine.

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 *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. The interpreter notifies the scheduler that the interpretation is completed.

[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.

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 [9] and C# [2] languages have been used to structure the language definition into manageable parts.

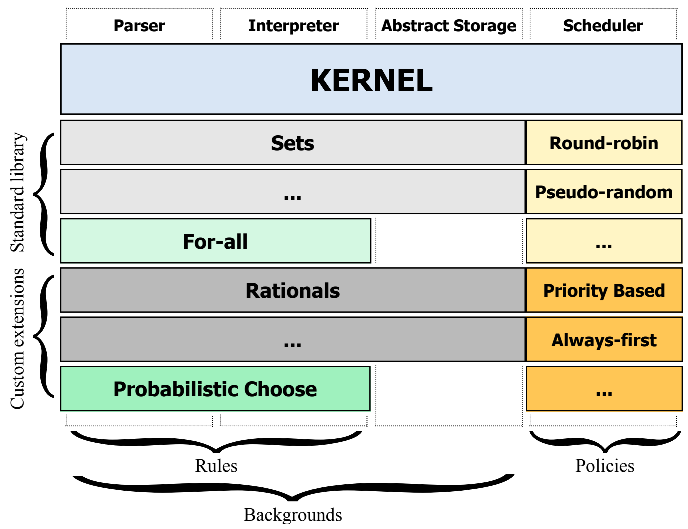

Fig. 3. Layers and Modules of the CoreASM Engine.

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.

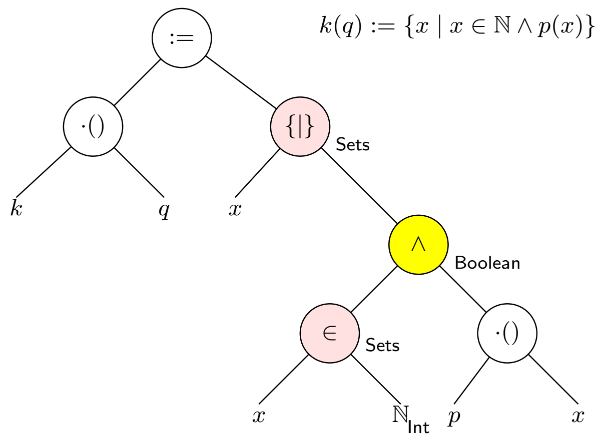

Fig. 4. Sample Annotated Parse Tree.

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 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 als 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 execute[5]. 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

[5] The 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.

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 [4]), 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.

4 An ASM specification for CoreASM

In this section we present a high-level specification of how the CoreASM engine performs one step of the simulated machine.[6] 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*)

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

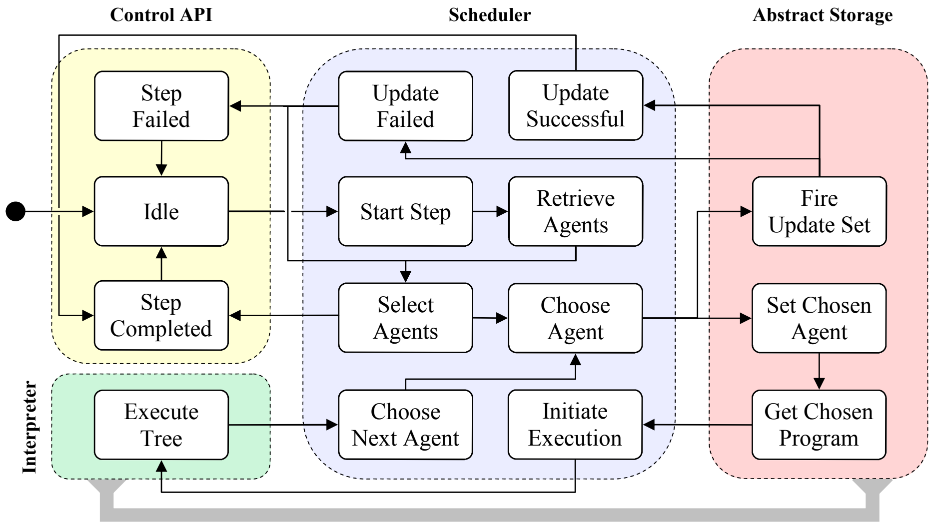

Fig. 5. Lifecycle of a STEP command.

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* in the simulated machine. We model the query process through the abstract macro GetValue which takes a location and a destination variable and assigns the value retrieved from the simulated state to the given variable. We use the notation $\ll$term$\gg$ 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 $\equiv$
  GetValue($agentSet$, (≪agents≫, ()))
  $selectedAgentSet := undef$
  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.

Abstract Storage

SetChosenAgent $\equiv$
  SetValue((≪self≫, ()), $chosenAgent$)
  $chosenProgram := undef$
  Next($stGetChosenProgram$)

GetChosenProgram $\equiv$
  GetValue($chosenProgram$, (≪program≫, (≪self≫)))
  Next($scInitiateExecution$)

Following the footsteps of [4], 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).

Scheduler

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. For a comprehensive specification of the interpreter, the reader is referred to [5]. 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 (not detailed here) 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).

Interpreter

ExecuteTree $\equiv$
  if $value(pos) = undef$ **then**
    if $plugin(pos) \neq undef$ **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.

Abstract Storage

FireUpdateSet $\equiv$
  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.

Scheduler

SuccessfulUpdate $\equiv$
  Next($caStepCompleted$)

UpdateFailed $\equiv$
  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($caIdle$)

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.

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. The ASM ground model method as a foundation for requirements engineering. In *Verification: Theory and Practice*, pages 145–160, 2003.
2. 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).
3. 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.
4. E. Börger and R. Stärk. *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
5. R. Farahbod, V. Gervasi, and U. Glässer. Design and Specification of the CoreASM Execution Engine. Technical Report SFU-CMPT-TR-2005-02, Simon Fraser University, February 2005.
6. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
7. ITU-T Recommendation Z.100 Annex F (11/00). *SDL Formal Semantics Definition*. International Telecommunication Union, 2001.
8. Microsoft FSE Group. *The Abstract State Machine Language*. cited June 2003, `http://research.microsoft.com/fse/asml/`.
9. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.