Executable Formal Specifications of Complex Distributed Systems with CoreASM

Roozbeh Farahbod[a], Vincenzo Gervasi[b], Uwe Glässer[c]

[a]*Defence R&D Canada—Valcartier, QC, Canada*
[b]*Dipartimento di Informatica, University of Pisa, Italy*
[c]*School of Computing Science, Simon Fraser University, BC, Canada*

Abstract

Formal specifications play a crucial role in the design of reliable, complex software systems. Executable formal specifications allow the designer to attain early validation and verification of design using static analysis techniques and accurate simulation of the runtime behavior of the system-to-be. With increasing complexity of software-intensive computer-based systems and the challenges of validation and verification of abstract software models prior to coding, the need for interactive software tools supporting executable formal specifications is even more evident. In this paper we discuss how CoreASM, an environment for writing and running executable specifications according to the ASM method, provides flexibility and manages complexity by using an innovative extensible language architecture.

Keywords:
Formal methods; Specification and modeling environment; Extensible language; Abstract state machines; CoreASM.

1. Introduction

Computer-based systems are increasingly integrated into our day-to-day life. They either control or provide platforms for our communication networks, transportation facilities, economic markets, health-care systems, and safety and security facilities. With the increasing complexity of these systems, efficient design and development of high quality computational systems that faithfully conform to their requirements are extremely challenging and the costs of design flaws and system failures are high. Proper understanding of the requirements, precisely documenting design decisions, and effectively

communicating such decisions with the domain experts as early as possible play important roles in the design of complex systems. These challenges call for adoption of proper engineering methods and tools and have motivated the use of *formal methods* in software engineering.

Abstraction and formalization provide effective instruments for establishing critical system requirements by precisely modeling systems prior to construction so that one can analyze and reason about specifications and design choices and better understand their implications [1]. There are many approaches to formal modeling of software and hardware systems. The *Abstract State Machine (ASM)* framework [2] has been used for computational and mathematical modeling of complex distributed systems with an orientation toward practical applications. The ASM framework offers a universal model of computation and serves as an effective instrument for analyzing and reasoning about complex semantic properties of discrete dynamic systems. For almost two decades now, abstract state machines have been studied, practiced, and applied in modeling and specification of systems to bridge the gap between formal and pragmatic approaches.

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

There is a variety of tools and executable languages available for ASMs, each coming with their own strengths and limitations. In [3], we propose a set of characteristic features (not further discussed here) that any tool environment for modeling and analysis of complex distributed systems should ideally support, and link these features to methodical aspects in the design of CoreASM (`www.coreasm.org`), a language and associate toolset for writing and executing ASM specifications. The description in [3] does however not address in any detail the fundamental technical concepts that form the foundation of the specification, design and implementation of CoreASM. In contrast, we explain here the key technical aspects of the tool architecture in detail and report on our experiences with design, implementation and use of CoreASM, building on its original formal semantic specification [4], which was developed prior to any coding as a blueprint for construction. CoreASM builds on sound theoretical foundations and existing development infrastruc-

ture, providing an innovative architecture for an extensible language. The system has been used both in research and in industrial contexts, supporting rapid experimentation as well as in-depth validation of complex software products.

The rest of this paper is organized as follows. Section 2 briefly describes the formal background of the ASM method; Section 3 presents the specification of the CoreASM language and the design of the CoreASM toolset, which is the main subject of this paper. This is followed in Section 4 by a description of the implementation of the toolset, with a special focus on its extensibility features and environment support. Section 5 reports on noteworthy experiences where the toolset has been applied to (and extended to address) practical problems. A short survey of related work and some conclusions complete the paper.

2. Abstract State Machines

This section briefly outlines the basic concepts of the ASM method for high-level design and analysis of distributed systems. For further details, we refer to [2, 5, 6]. ASMs offer a versatile mathematical method for modeling of discrete dynamic systems with the goal to bridge the gap between computation models and specification methods. ASM models essentially are rigorously defined *pseudocode* programs operating on abstract data structures [2]. Building on common notions from discrete mathematics and computational logic, static and dynamic aspects of systems are modeled at any desired level of abstraction by combining two concepts of *abstract states* and *transition systems*. ASMs have been used in modeling of architectures, languages, protocols and virtually all kinds of sequential, parallel and distributed systems with a notable orientation toward practical applications [2, 7, 8, 9, 10, 11].

2.1. ASM systems engineering method

The ASM method aims at industrial system design and development by integrating precise high-level, problem-domain oriented modeling into the design and development cycle, and by systematically linking abstract models down to executable code.

The method consists of three essential elements: *a)* capturing the requirements into a precise yet abstract operational model, called a *ground model* ASM, *b)* systematic and incremental refinement of the ground model down to

the implementation, and *c)* experimental model validation through simulation or testing at each level of abstraction. This process emphasizes freedom of abstraction as a guiding principle, meaning that original ideas behind the design of a system can be expressed in a direct and intuitive way so as to enable system designers to stress on the essential aspects of design rather than encoding insignificant details. To this end, it is vital that the method allows for, and actually encourages, the use of notational conventions, for instance, as in the typical phrase "In the following, we use the notation ¡notation¿ to mean ¡semantics¿". It is also understood that authors can use the full extent of mathematics and computer science notations if that is instrumental to express themselves clearly. Any executable implementation must thus allow for similar extensibility, which constitutes a significant design challenge in itself.

Starting from a ground model and applying the process of step-wise refinement [12], a hierarchy of intermediate models can be created that are systematically linked down to the implementation. At each step, the refined model can be validated and verified to be a correct implementation of the abstract model. The resulting hierarchy serves as a design documentation and allows one to trace requirements down to the implementation.

2.2. Distributed ASMs

The original notion of *basic ASM* was defined to formalize simultaneous parallel actions of a single computational agent. A basic ASM M is defined as a tuple of the form $(\Sigma,\ \mathcal{I},\ \mathcal{R},\ P_M)$ where Σ is a finite set of function names and symbols, $\mathcal{I}$ is a set of initial states for Σ, $\mathcal{R}$ is a set of transition rule declarations, and $P_M \in \mathcal{R}$ is a distinguished rule, called the *main rule* or the Program of machine M.

A state $\mathfrak{A}$ for Σ is a non-empty set X together with an interpretation $f^{\mathfrak{A}} : X^n \mapsto X$ for each function name f in Σ. Functions can be *static* or *dynamic*. Interpretations of dynamic functions can change from state to state. Pairs of a function name f and an optional argument $(v_1, \ldots, v_n)$ are called *locations*. The evaluation of a transition rule in a given state produces a finite set of *updates* of the form $\langle (f, \langle a_1, \ldots, a_n \rangle), v \rangle$ where f is an n-ary function name in Σ and $a_1, \ldots, a_n, v \in X$. An update (f, args, v) prescribes a change to the content of location $f(\mathit{args})$ taking effect in the next state.

A distributed ASM (DASM) M_D is defined by a dynamic set AGENT of autonomously operating computational *agents*, each executing a basic ASM. This set may change dynamically over runs of M_D, as required to model

a varying number of computational resources. Agents of M_D interact with one another, and typically also with the operational environment of M_D, by reading and writing shared locations of a global machine state. The underlying semantic model resolves potential conflicts according to the definition of *partially ordered runs* [9, 5].

M_D interacts with its operational environment—the part of the external world visible to M_D—through actions/events observable at external interfaces, formally represented by controlled and monitored functions. Of particular interest are *monitored functions*, read-only functions controlled by the environment. A typical example is the abstract representation of global system time in terms of a monitored function *now* taking values in a linearly ordered domain TIME. Values of *now* increase monotonically over runs of M_D.

3. Specification and Design of CoreASM

More than a full year of work went into the formal specification and design of the CoreASM architecture, its language and the simulation engine prior to any coding. The resulting formal model [4], defined in terms of an abstract state machine, served as a precise blueprint for requirement analysis and for reasoning about design decisions. The formal approach greatly simplified the analytical validation of the design of the core components and their interoperability. Starting from the foundation laid in [4], the original specification of CoreASM, the formal model turned out to be stable and robust so that it evolved only marginally during the actual development and implementation phase. In fact, all of the essential design concepts established in the original model turned out to be sensible choices and are virtually identical in the most recent version documented in [13]. This section illustrates in detail the formal requirements and design specification of the CoreASM language and tool architecture. For a complete description of the formal CoreASM model we refer to [13].

Since the CoreASM language definition and underlying semantics are virtually identical to the ASM language, CoreASM directly inherits some of the prominent features of the ASM modeling framework. As explained in Section 2, ASM models are rigorously defined pseudocode programs operating on abstract data structures, a concept that supports writing of concise and intelligible specifications with a precise semantic foundation. Nonetheless,

abstract machine operations and data structures can be fairly high-level[1] and yet in principle be executable. The ASM framework comes with a sound and powerful notion of stepwise refinement [12] that helps coping with complexity by structuring the design of a system into suitable levels of abstraction and linking them down to a concrete model. The design of both distributed and parallel systems is supported by providing asynchronous and synchronous computation models. Explicit and implicit non-determinism offer flexible ways of avoiding insignificant details and modeling interaction with the external world.

The latest version of the CoreASM toolset offers *a)* an extensible specification language, *b)* an extensible multi-agent ASM simulation engine, faithful to the mathematical definition of ASMs, that animates CoreASM specifications in addition to providing other services, such as parsing, through a rich API, *c)* a library of optional plugins offering additional features and language constructs not originally part of the ASM dialect, and *d)* an Eclipse front-end with dynamic syntax high-lighting and a command-line user interface.

3.1. From ASM to CoreASM

This section illustrates a simple CoreASM model and in part also its ASM equivalent using the railroad crossing example of [13] which is based on the specification of a railroad crossing gate controller published in [14].

A system controls a gate at a railroad crossing with multiple tracks on which trains can travel in both directions. Sensors on the tracks can detect if a train is *coming* or if it is currently *crossing*. The gate is controlled by two signals *open* and *close*. The purpose of the system is to keep the gate closed if a train is crossing (safety) and to keep it open otherwise (liveness).

The Rail Road Crossing ASM consists of two basic abstract machines, TrackControl and GateControl, respectively controlling the tracks—sending signals to the gate controller—and maintaining the state of the gate—opening or closing the gate in response to gate signals. The environment sets the value of the function *trackStatus* based on the track sensors data.

```
function trackStatus : Track -> {empty, coming, crossing}
function gateState : -> {opened, closed}
```

There is an implicit *deadline* associated with every track t, indicating the maximum available time, with regard to track t, to safely close the gate.

[1] Arbitrary structures can be used to reflect the underlying notion of state [2, p. 22].

```
function deadline : Track -> TIME
```

The track control program TrackControl is a parallel combination of two ASM rules: 1) closing the gate if needed (for all tracks, calculate new deadlines, send a close signal if needed, and clear passed deadlines); 2) opening the gate if it is safe to do so. The program is defined as shown in Figure 1.

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

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

TrackControl ≡
  forall t **in** TRACK **do**
    SetDeadline(t)
    SignalClose(t)
    ClearDeadline(t)
  SignalOpen

SetDeadline(x) ≡
  if $trackStatus(x) = coming$ **and** $deadline(x) = \infty$ **then**
    $deadline(x) := now + waitTime$

Figure 1: CoreASM version of TrackControl with SetDeadline(x) and its ASM equivalent

SignalOpen, for example, is defined as follows.

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

The predicate *safeToOpen*, used in the SignalOpen rule, is defined as follows.

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

where d_{open} refers to the time it takes to actually open the gate. This definition translates into CoreASM as shown below, in which the keyword "derived" denotes the definition of a function whose value is dynamically calculated at runtime.

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

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

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

3.2. CoreASM Formal Specification

We firmly belief that the design and development of a reliable framework for mathematical system modeling ought to start with a formal specification of its language and tool architecture.[2] Building on a formal model serves practical needs in the design and development process, as will be explained.

The CoreASM language—its abstract syntax and underlying semantics—is specified in terms of an interpreter in the form of an abstract state machine, thereby ensuring executability of the language together with providing its formal semantics. Syntactical patterns and their corresponding semantics are defined using the following notation.

$$(\!| \mathit{pattern} |\!) \rightarrow \quad \mathit{actions}$$

Such an expresion can be directly mapped to a rule of the form

$$\textbf{if } \mathit{conditions} \textbf{ then } \mathit{actions}$$

where the *conditions* are derived from the pattern. For instance, the CoreASM assignment rule is defined as follows.

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

The notation we use here has been introduced in [4]. It will suffice to say that the semantics is given by ASM rules guarded by syntactical patterns; a variable *pos* indicates the subtree which is being evaluated, and is used to navigate the syntax tree. Inside a pattern, a generic parse tree node is denoted with $\boxed{?}$, regardless of being evaluated or not (while an empty box indicates an unevaluated node). Prefix superscripts name locations. In the ASM rules, each symbol is bound to the corresponding value in the pattern.

[2] How can one convincingly argue for genuine benefits of using a formal framework for analysis and specification of systems otherwise?

Evaluation of ASM rules results in assigning a triple (location,updates,value) to the evaluated node; this operation is denoted as $[\![pos]\!] := (l, u, v)$. The idea here is that for each individual program, the CoreASM parser generates an annotated parse tree as input for the interpreter. Each node of a parse tree potentially has a reference to the plugin that defines the corresponding syntax. By traversing a parse tree, the interpreter generates a multiset of *update instructions*, each of which represents either an update, or an arbitrary instruction to be processed at a later stage by means of plug-ins that generate the actual updates. In the above expression *pos* refers to the node in the parse tree that is being evaluated, and α and β respectively refer to the *lhs* and *rhs* nodes in the parse tree. Control proceeds from node to node either by explicitly assigning values to *pos*, or by evaluating *pos*, i.e. setting a triplet of (*locaction*, *updates*, *value*) to $[\![pos]\!]$.

The above rule does not syntactically constrain assignment to be performed exclusively to variables or functions, rather any expression that can be evaluated to a modifiable location is syntactically acceptable in the *lhs* of an assignment; likewise, any expression that can be evaluated to a value is acceptable in the *rhs* of an assignment.

The design of the CoreASM simulation engine and its architecture is specified using Extensible Control State ASMs (eCASMs). Figure 2 illustrated an example [13]. Upon receiving a load command the engine loads a new specification in the following consecutive steps. It first clears previously loaded data, then reads the specification file and parses the specification header to get the list of specific plugins required to be loaded. Loading the required plugins is done in two steps. First, all the package plugins (plugins that are basically a set of other plugins) are expanded and their enclosed plugins are added to the list of required plugins. In the next step, plugins are loaded one by one according to their loading priority. After all the required plugins are loaded, the specification is parsed using the grammar rules provided by the plugins. To prepare the engine for the first simulation step, Abstract Storage is initialized taking into account all plugins contributions, such as backgrounds, universes, functions, and macro rules. The concept of *background* [15] refers to an implicitly given part of an abstract machine state, assuming that it provides whatever standard means are normally supposed to be available in a given application context.[3] Further, a universe of Agents

[3] A realistic description of algorithms involves quite a rich background, including num-

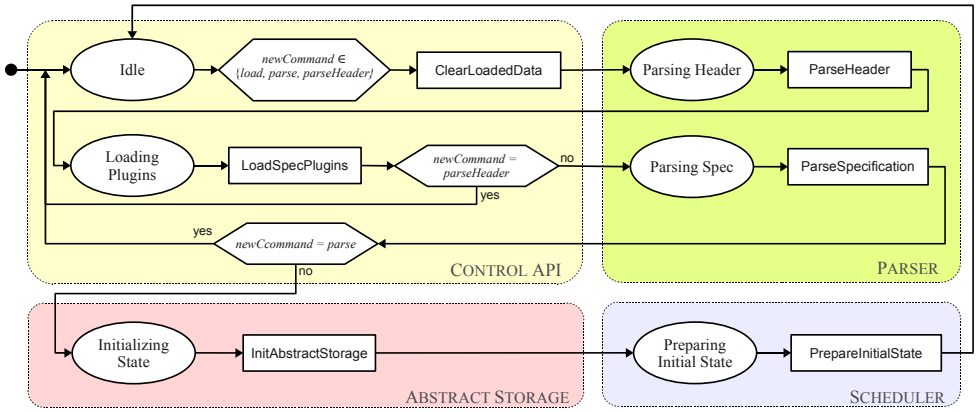


Figure 2: Control State ASM of loading a specification in CoreASM

and a function *program* that assigns programs to agents are also created in this step. Finally, an initial state is created with at least one agent that, in the first step of the simulation, will run the main program.

Control State ASMs (CASMs) [2] are a practical class of abstract state machines with an easy-to-understand graphical representation. Based on CASMs, we define eCASMs [17] as a class of control state ASMs the behavior of which can be arbitrarily extended using plugins. In the following section we look into the extensibility concept of CoreASM.

3.3. An Open and Extensible Framework

Abstract state machines are used in diverse application domains, some of which require the introduction of special rule forms and data structures. Hence, CoreASM is implemented to be flexibly extensible by third parties to meet application specific requirements. Furthermore, supporting freedom of experimentation, we would like to allow various modeling tools and environments to closely interact with the engine and also allow researchers to use variations of the engine. The design of a *plugin-based architecture*, based on a minimal kernel for the CoreASM language and modeling environment, offers the extensibility of both the language and its simulation engine. A micro-kernel that forms the *core* of the language and the engine contains the bare

bers, sets, multisets, maps, sequences, and the like, since all these things are generally taken to be available when designing algorithms [16].

essentials, that is, all that is needed to run only the most basic ASM. Most of the constructs of the language and operation of the engine come in the form of plugins extending the kernel. This extensibility concept is explored in detail in [17, 13].

3.3.1. Extensible Language

Language extensibility is not a new concept [18]. For instance, there are a number of programming languages that support some form of extensibility, ranging from introducing new macros to the definition of new syntactical structures. CoreASM, however, offers the possibility of extending and modifying both the syntax and semantics of the language, keeping only the bare essential parts invariable. Plugins thus require extending the grammar of the core language by providing new grammar rules together with their semantics. As a result, every time a CoreASM specification is being loaded, based on the set of plugins that the specification uses, the engine builds a language and a parser for that language to parse the specification. Since the set of all the possible plugins and their grammar rules is not known at design time (due to the plugin-based architecture), one of the challenges was to equip the engine with a fast parser generator capable of generating parsers with look-ahead of more than one to allow the co-existence of multiple grammar rules all starting with the same pattern.

3.3.2. Extensible Engine

Serving distinct practical needs, there are two different mechanisms for extending the CoreASM engine. Plugins can extend and/or modify the functionality of specific engine components (like the parser or the agent scheduler) either by introducing additional data or behavior to those components or they can extend the control state ASM of the engine by interposing their own code in between state transitions of the engine. The latter mechanism enables a wide range of extensions of the engine's execution cycle for the purpose of implementing various practically relevant features, such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the run-time behavior of the simulated machine, e.g., through coverage analysis, profiling and the like.

The eCASM of the engine associates an *extension point* with each state transition. Plugins can extend the engine's control state ASM by registering for these extension points. At any extension point, if there is any plugin registered for that point, the code contributed by the plugin for that transi-

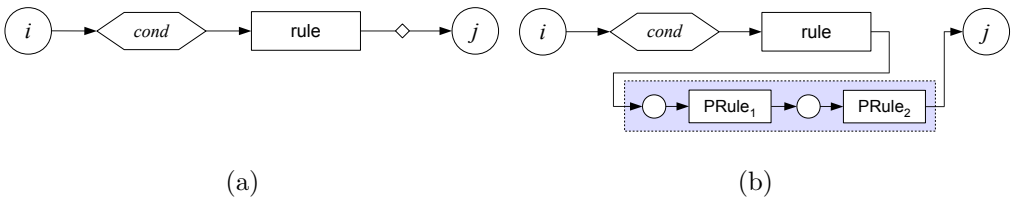


Figure 3: (a) An Extensible Control State ASM and (b) its extended form

tion is executed before the engine proceeds to the next control state. As an example, the eCASM of Figure 3(a) can be executed with a set of extension point plugins $\{p_1, p_2\}$ contributing rules $PRule_1$ and $PRule_2$ that (potentially) extend the execution of the machine to the control state ASM of Figure 3(b).

A plugin, for example, may monitor the updates that are generated by a step—possibly checking conditions on these updates—before they are applied to the current state of the simulated machine, thus implementing a kind of *watch-point* to suspend execution of the engine when certain updates are generated, which is useful for debugging purposes.

Finally, the set of plugins used in a specification is determined by the specification itself, so that different users of CoreASM can adapt the language syntax and (within reason) semantics to their own specific needs, while at the same time relying on a solid formal background that guarantees a consistent behaviour without constraining notational convenience.

3.3.3. Open Framework

CoreASM is an open framework meant to be used with complementary tools, e.g., for symbolic model checking and automated test generation. To do this, the CoreASM engine comes with a sophisticated and well defined control API serving for various operations such as loading CoreASM specifications, starting an ASM run, or performing single execution steps of the simulated machines. Aiming at a platform-independent implementation of CoreASM that supports future improvement and modifications as needed by its growing user community, the whole framework is implemented in Java under the Academic Free License version 3.0 (AFL 3.0). AFL 3.0 is an open source license with no reciprocal obligation to disclose source code; it provides a good compromise between public availability of the original source code and the existence of proprietary editions and extensions for commercial applications.

4. Implementing CoreASM

Following the formal specification and design of the CoreASM architecture, implementing the first prototype (including the CoreASM kernel and its basic plugins) was rather straight-forward and took only about six man months in total by two graduate students. One of the main challenges was to properly adapt a parser generator that is fast enough to generate parsers from fragments of grammar rules provided by different plugins every time a CoreASM specification is loaded.

The CoreASM source code, with about 59K lines of code, is available on Sourceforge (`coreasm.sf.net`). Since its first beta release in September 2006, CoreASM went through a number of revisions. Its latest version (currently under testing) offers substantial improvements over its previous versions in terms of features and performance (see `wiki.coreasm.org` for details).

4.1. The Architecture

Closely following the design of the engine, the Java implementation of CoreASM implements the kernel of the engine in terms of four components and a Control API. The interface of the components are defined by four Java interface files: `Parser`, `Interpreter`, `AbstractStorage`, and `Scheduler`. For every component, a default implementation is provided in the form of a Java class file. Since Control API acts as a double interface, providing services both to the environment of the engine and to its internal components (the former being a subset of the latter), two Java interface files together define the interface of the engine: (i) a `CoreASMEngine` interface defines the interface of the engine to its outside environment offering services such as loading, parsing, or execution of specifications; (ii) a `ControlAPI` which extends the `CoreASMEngine` interface providing access to every component, a mapping of plugin names to actual plugin instances, and error reporting services. An implementation of the CoreASM engine is provided by the Java class file `Engine` which implements the `ControlAPI` interface.

The `CoreASMEngine` interface provides a comprehensive interface to the engine. Through this interface, applications can *a)* load and execute CoreASM specifications and access the (simulated) state of the machine at runtime, *b)* use the engine to parse CoreASM specifications into parse-trees (which can be externally processed for various purposes such as model checking [19] or pretty printing), and *c)* modify various engine properties as well as monitoring the behavior of the engine at runtime.

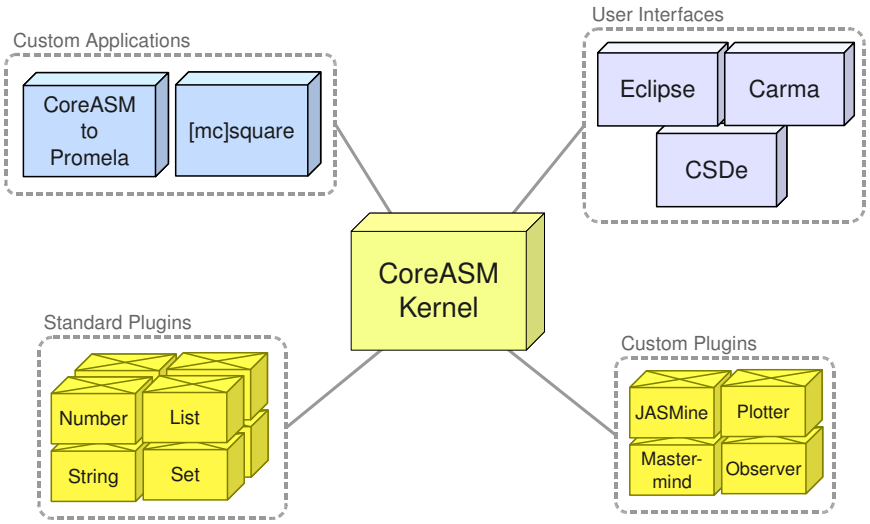


Figure 4: CoreASM Kernel, Plugins, and Applications

There are currently two user interfaces available for CoreASM (see Figure 4): a comprehensive command-line user interface, called Carma, and a graphical interactive development environment in the Eclipse platform, known as the CoreASM Eclipse Plugin. There is also a sophisticated tool under development for creating and modifying Control State ASMs and translating them into CoreASM specifications, called CSDe.

The CoreASM kernel also defines the skeleton of a CoreASM plugin in form of a Java abstract class `Plugin`. Various types of extensions that plugins can provide to the engine (see [17] for a complete list) are defined in terms of Java interface files. Every CoreASM plugin must extend the `Plugin` abstract class and typically implement one or more of the extension interfaces.

4.2. The CoreASM Engine

CoreASM engine is represented by the `CoreASMEngine` interface and is implemented by the `Engine` class file which serves two purposes: (i) it provides an implementation for the interface of the engine to its outside environment, and (ii) it acts as a container for the main components of the engine and maintains the control state of the CoreASM engine. In order for the engine to be always responsive to its environment, the `Engine` object runs in two parallel processing threads: one, being the environment or the *caller*'s thread,

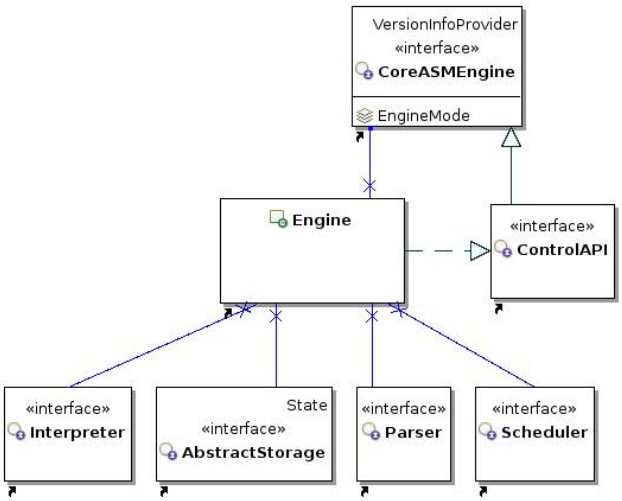


Figure 5: Components of the CoreASM Engine

responds to requests from the environment (such as sending commands, setting engine properties, or retrieving updates) and the other one maintains the internal control flow of the engine.

4.2.1. Abstract Storage

The Abstract Storage, one of four components forming the CoreASM kernel, maintains a representation of the current state of the simulated machine in CoreASM, and provides interfaces to retrieve and update values assigned to the locations of the simulated state. This component is implemented by more than three dozen classes in the package `org.coreasm.engine.absstorage`. A hierarchy of classes implement various types of elements defined in the kernel (see Figure 6). At the root of this hierarchy, we have the `Element` class which is the superclass of all the *values* in CoreASM states, implementing the ELEMENT domain. Following the CoreASM specification, every instance of `Element` has a background and a notion of equality. Three immediate subclasses `BooleanElement`, `RuleElement`, and `FunctionElement` respectively implement the domains of Booleans, transition rules, and state functions.

4.2.2. The Parser

Implementing the parser component of the CoreASM engine was quite a challenge. We were looking for fast and efficient parser generators that can be called upon loading a specification to generate a parser based on the

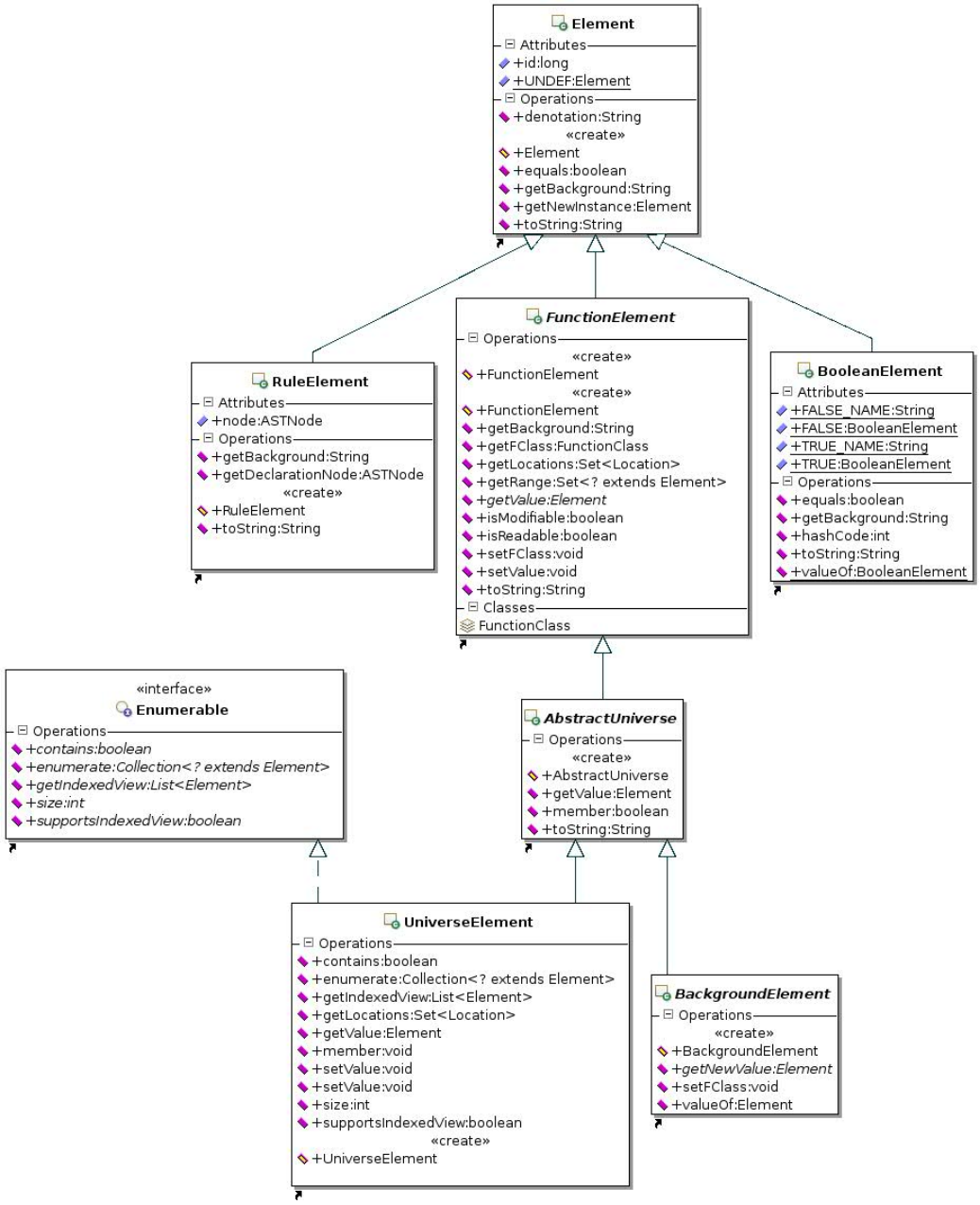


Figure 6: Core Elements Defined in the Abstract Storage

grammar provided by the specific plugins that are used in that specification. We looked into a number of available open source parser generators in search of an efficient LL(k) parser generator written in Java and we eventually found jparsec (`jparsec.codehaus.org`), a recursive-descent parser combinator framework written for Java. In contrast to traditional parser generators like YACC or ANTLR, jparsec grammar is written in native Java language and is defined in terms of special Java instances of a `Parser` class. Each parser object represents a grammar rule and can be combined with other parser objects to create more complex production rules.

This feature of jparsec appeared to be very beneficial for CoreASM. Upon loading a specification, the kernel provides references to the core parser objects (such as white spaces, identifiers, terms, etc.)[4] and makes them available for plugins to build upon. Plugins in turn provide their contributions to the parser in form of new jparsec parser objects. The kernel then combines all these parser contributions together to create the final parser that will be used to parse the specification.

4.2.3. CoreASM Plugins

Every CoreASM plugin must extend the abstract class `Plugin` and most likely implements at least one of the nine plugin interfaces offered by the engine [17]. A CoreASM plugin is usually accompanied by a number of auxiliary Java classes. As a result, every CoreASM plugin is expected to be packed into a single JAR file or a folder of its own together with an identification file. When an instance of `Engine` is initialized, it searches the specified plugin folders, creates a catalog of available plugins so that they can be later instantiated if needed. As a result, to add a new plugin to CoreASM, one only needs to put the compiled class files of the plugin together with an identification file into a plugin folder of the engine.

4.3. User Interfaces and Tools

The CoreASM engine is implemented as a Java component and requires a *driver* program (such as a user interface) to run the engine, e.g., to pass specification files to the engine and to control its simulation run by manipulating parameters. Here we briefly present the currently available drivers and user interfaces for CoreASM.

[4]Some of these core parsers, such as the one for parsing CoreASM terms, can also be extended by plugins.

4.3.1. Carma

Carma is a comprehensive command-line user interface for CoreASM that offers rich control over the runs of the engine through a number of command-line options and switches. To execute a specification, users can simply run Carma on the command line and pass it the name of the specification file[5] as an argument. By default, an ASM run managed by Carma runs indefinitely (which is according to the theoretical definition of a run), but the tool offers a number of termination conditions, such as termination after a number of steps, termination on empty updates, and termination when there is no valid agent with a defined program, which can be selected by the user. As an example, the following command runs the CoreASM specification `MySpec.coreasm` using Carma and stops after 30 steps or after a step that generates empty updates; it also provides a print-out of the final state before termination.

```
carma --steps 30 --empty-updates --dump-final-state MySpec.coreasm
```

4.3.2. The CoreASM Eclipse Plugin

The CoreASM Eclipse Plugin is a graphical interactive development environment for CoreASM in form of a plugin for the well-known Eclipse software development platform. The IDE provides various options to control execution of CoreASM specifications. The plugin extends the Eclipse platform to support dynamic syntax highlighting (using Eclipse's native framework) and interactive execution of CoreASM specifications. Since the language of CoreASM for a given specification is defined by the set of plugins used by that specification, with every change to the specification, the editor component of the CoreASM Eclipse Plugin passes the specification to the CoreASM engine and gets the set of plugins that are used by the specification. The editor then asks the plugins for the set of keywords, functions, universes and backgrounds they provide and uses this information to offer a dynamic syntax highlighting of the specification.

Figure 7(a) shows a snapshot of the CoreASM environment in Eclipse. At the top left corner ①, the toolbar is extended to include buttons to pause, resume and stop a simulation run. The editor ② provides dynamic syntax highlighting for CoreASM specifications based on the set of CoreASM plugins used in the specification. A configurable output console ③ provides a print-

[5]The specification can be a text file, as is traditionally the case, or an OpenOffice document with commentary intertwined with specification text, in the spirit of literate programming.

out of the results of the simulation with optional additional information on the simulation process and the state of the simulated machine.

4.3.3. CSDe

The Control State Diagram editor (CSDe), under development by Piper J. Jackson [20], is a sophisticated tool for creating and modifying Control State ASMs and translating them into CoreASM specifications. The tool is implemented as a plugin for the Eclipse software development platform. The plugin allows the user to work with Control State Diagrams (CSDs) using a point-and-click schema (see Figure 7(b)).

The simplicity of control state diagrams and the intuitiveness of the graphical user interface work together to allow users to confidently contribute to the design, regardless of their technical background. The diagram editor (CSDe) is capable of automatically transforming diagrams into CoreASM specifications. Since control state diagrams do not necessarily include initial states of the system or other more concrete information required for machine execution, such specifications may not be directly executable. However, they provide an abstract structure for the design of systems and act as foundations for further development of the specifications. The automatic translation feature facilitates the transition from high-level design ideas expressed in graphical form towards less abstract specifications.

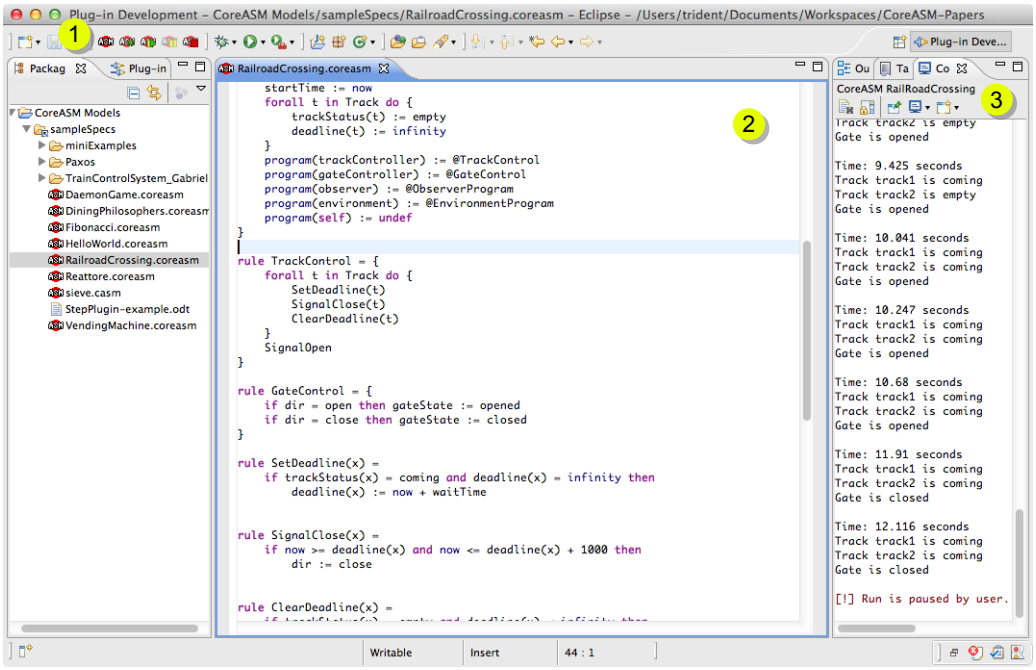
CSDe, in its current form, is primarily a proof of concept. As an Eclipse plugin, it relies on older versions of Eclipse and the Eclipse Graphical Modelling Framework (GMF) and does not work with the latest versions of the framework.

4.4. Using CoreASM: An Example

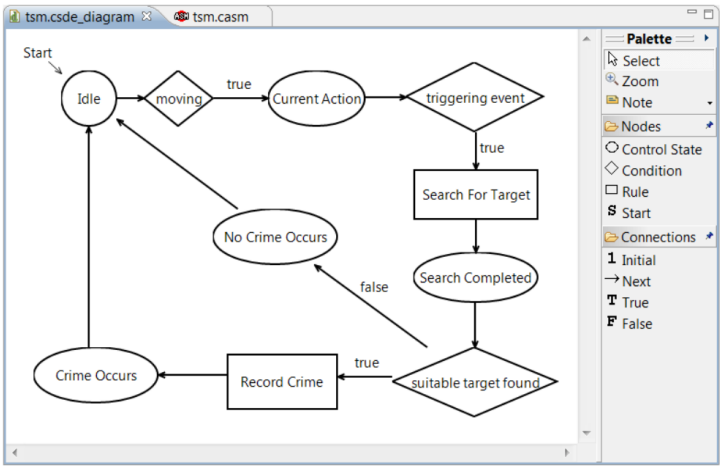
In this section, we briefly go through the process of installing CoreASM and running one of the sample specifications that are available on the CoreASM website.

4.4.1. Installing CoreASM

As mentioned earlier, there are two font-ends for CoreASM: the command-line Carma interface and the graphical CoreASM Eclipse plugin. Both packages include the latest version of the CoreASM engine. Carma can be downloaded as a ZIP package from the CoreASM website and will be executable on any system running Linux, Windows, or Mac OS X with Java Runtime



(a) CoreASM Eclipse Plugin



(b) CSDe: A Control State Diagram editor for CoreASM

Figure 7: CoreASM Tools in Eclipse

Environment version 1.6 or higher installed. The latest version of the CoreASM Eclipse plugin can be installed from within the Eclipse environment by pointing to the CoreASM Eclipse update site. Instructions in detail are available on www.coreasm.org/download.

4.4.2. Running the Railroad Crossing Example

Running the CoreASM model of the Railroad Crossing Example of Section 3.1 allows us to validate the behavior of the gate controller[6]. For the sake of brevity, here we focus on using Carma to run this specification.

The execution provides a printout of the states of the system. The output shows that the controller keeps the gate open while there is no train on the tracks and keeps it closed as long as there is at least one train crossing the intersection. Since there is no termination condition and the gate controller can run indefinitely, we can limit the run by providing the maximum number of computation steps the simulation should run for. This is done by passing the number of steps to Carma using the `-s` (or `--steps`) argument:

```
$ carma -s 100 RailroadCrossing.coreasm
```

4.4.3. Using the Observer Plugin

It is sometimes desirable to have a machine-readable log of the execution of a specification for offline analysis and visualization. Such a feature allows for a clear separation of the execution and the analysis. To offer this feature, the Observer plugin can be used to monitor the execution of specifications in CoreASM and produce an XML log of the updates that are produced after every computation step. The plugin can be configured so that only the updates on certain locations of interest are recorded. For example, the following line in our specification configures the Observer plugin to monitor only the updates that affect the states of the gate and tracks:

```
option Observer.LocationsOfInterest "trackStatus gateState"
```

For this particular example, we have developed a visualizer that loads an XML log of a simulation (automatically produced by the Observer plugin) and provides a visual account of the simulation. Figure 8 presents a screenshot of this visualizer.

[6] See http://coreasm.org/publications/scp-rce.pdf for the full specification.

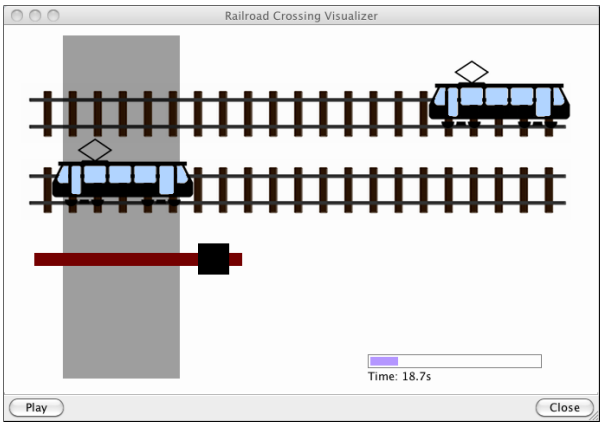


Figure 8: Visualizing a Run of the Railroad Crossing Specification

5. CoreASM Applications

CoreASM has been put to the test in a range of applications in the private and public sectors, spanning computational criminology,[7] coastal surveillance, decision support, Web services, and high-level synthesis in hardware design. The diversity of application fields has been invaluable to examine the practicability of using CoreASM for requirements analysis, design specification and rapid prototyping of abstract executable models. In this section, we briefly discuss some of these projects. Table 1 provides examples of the specification sizes and CoreASM features used in three of these projects.

Dynamic Resource Configuration & Management Architecture (DRCMA) is a highly adaptive and auto-configurable, multi-layer network architecture for distributed information fusion [21]. The ASM model of the underlying design provides a reliable basis for reasoning about key system attributes at an intuitive level of understanding, supporting requirements specification, design analysis and validation of dynamic properties. Building an abstract yet executable DRCMA model in CoreASM enabled advanced experiments to validate consequential design decisions at a fairly high levels of abstraction.

Our work on integrating ASM modeling with Interpreted Systems for situation analysis decision support system design [22], exemplifies the benefits of using ASM and CoreASM in combination with the Interpreted Systems

[7] Computational criminology is a rapidly growing field that explores the use of computer science methods and tools in different stages of studying complex crime phenomena.

approach of [23] in modeling multiagent systems for situation analysis. Refinement of the abstract model into an executable CoreASM model serves two purposes: *a)* it helps finding ambiguities, missing pieces and loose ends of the model and forces the system analyst/modeler to think clearly about the main concepts and their definitions, and *b)* it supports experimental validation through execution (simulation). The outcome of this work initiated another project with Defence R&D Canada proposing a formal modeling framework for high-level design and analysis of Situation Analysis Decision Support Systems (SADSS) in which CoreASM serves as means to enable validation of abstract models with distributed and mobile components [24, 25]. This framework captures common concepts and notions of situation analysis and situation awareness, allows for reasoning about knowledge, uncertainty and belief change, and enables rapid prototyping of abstract executable decision support system models. Experimental study of SA scenarios as presented here can considerably enhance our insight into intricate system dynamics and simplify the challenging task of deriving meaningful conformance criteria for checking the validity of Situation Analysis Decision Suppor domain models against established operational concepts of Marine Safety & Security.

The Mastermind project [26] is a pioneering interdisciplinary project in computational criminology that focuses on modeling and simulation in the study of spatiotemporal patterns of offender behavior in urban environments. At the heart of Mastermind is a robust ASM ground model, developed over many iterations, for checking the validity of the model with respect to established crime theories. The process of establishing the key properties and ensuring the validity of the model was greatly facilitated by running experiments on abstract models using CoreASM. In this project, CoreASM has played an important role in facing the challenges of two major phases of the Mastermind project, namely *formalization* and *validation* [26].

Altenhofen and Börger [27] analyze a given cluster protocol implementation using an abstract ASM model, which they refine into an executable CoreASM model for running scenarios. Lemcke and Friesen at SAP Germany [28] propose a Web services composition algorithm for collaborative business processes defined in terms of a distributed ASM, using CoreASM for executing their ASM model to show that the generated orchestration steers the execution of the business processes as intended. Beckers et. al [29] use the simulation capabilities of CoreASM in their approach to model checking ASMs without the need for translation of the ASM specification into the modeling language of an existing model checker.

Project	Spec. Lines	Modules	DASM	JASMine	Custom Plugin	GUI
DRCMA	~1,100	6	✓	✓		✓
SADSS	~1,200	5	✓		✓	✓
Mastermind	~1,600	1	✓		✓	✓

Table 1: Examples of Specification Sizes and CoreASM Features Used

CoreASM's extensibility has played a crucial role in writing specifications efficiently. For example, in [30] CoreASM was used to build an executable model of the behavior of a liver cell, and of how it would react to changing environmental conditions. To inspect the health of the cell, we used a plugin to graphically plot an arbitrary number-valued function on the screen. Similarly, in [31] we produced a formal specification of the behavior of a web browser, for the purpose of validating properties of several web applications framework on the HTML 5 specification. Such a large specification would have been impossible to write without the ability of directly expressing domain-specific concepts such as parsing an HTML file into a Document Object Model, or parsing a Javascript source program into an abstract syntax tree (for the purpose of interpreting it).

In fact, we developed plugins to incorporate trees and grammars into the fabric of the language – a grammar, for example, is mapped to the ASM state in such a way that every non-terminal is a location in the state (i.e., a variable whose value is the corresponding production), and as such it can even be dynamically updated during the computation. By implementing extensions to the parser (defining additional grammar rules), the interpreter (defining additional semantics) and the vocabulary (defining additional constants and literals), the grammar plugin seamlessly incorporates into the language the ability to define grammar rules by assignment:

```
Op := "+" | "-"
Term := ID | Number | @Term . Op . @Term | "(" . @Term . ")"
```

where the operators `|` (alternative), `.` (sequence), and `@` (lazy evaluation for recursion), and the predefined constants `ID` and `Number` are provided by the plugin. The actual parsing of a string is then performed by a rule (also provided by the plugin) such as

```
parse "1+(test-2)" by Term into T
```

which results in the term T containing the parse tree for the given string.

6. Related Work

Over the years, a variety of executable ASM languages has been developed. The first generation of such tools goes back to basic interpreters and compilers written in `C` [32], Prolog [33] and Scheme [34]. Besides, theoretical frameworks emerged, such as a universal ASM for executing ASM models [35]. With more experience, a second generation of ASM tool environments was developed: Microsoft's *AsmL* [36] (which was integrated in the .NET initiative) and *Xasm* [37] use compilers, whereas the *ASM Workbench* [38], *AsmGofer* [39], and *Asmeta* [40] are interpreters.

The above languages build on predefined type concepts rather than the untyped language underlying the theoretical ASM model. The most prominent ASM tools are Asmeta, AsmL and Xasm. The Asmeta language implements all the constructs of basic, structured and multi-agent ASMs defined in [2] as a fully typed language with limited extensibility features. AsmL is a strongly typed language that also includes many object-oriented features and constructs for rapid prototyping of component-oriented software, thus departing from the theoretical ASM model; rather it comes with the richness of a fully fledged programming language. Most of these languages do not provide run-time system support for distributed ASM models (only Asmeta and AsmGofer provide some sort of support); only Xasm (and Asmeta in a limited form) allows systematic language extensions; however, the Xasm language itself diverts from the original definition of ASMs and seems closer to a programming language.

State-based formal methods that view the states of a system in terms of mathematical structures are common for practical system design and analysis. In addition to ASM, one can point to methods such as Alloy [41], B [42], CASL [43], the Vienna Development Method (VDM) [44], and the Z notation [45] as the most popular approaches that share many similar concepts and rely on tool support for analysis of specifications. In fact, the ingredients of the ASM method are not original. For example, The concept of abstract states are known from the theory of abstract data types and algebraic specifications, VDM, and Z, and the ASM refinement concept generalizes the method which has been introduced by Wirth [46] and Dijkstra [47] and has been adapted to numerous formal specification methods including Z and B. What is unique about ASM is the simplicity of the method and the freedom it offers the practitioner to choose an appropriate level of abstraction and a combination of concepts, notations and techniques that can be integrated

by the framework as elements of a uniform mathematical background [48]. Other well known design and computation models are naturally embedded into ASMs where they can be recognized by specializing the signature, the rules, the constraints, and the runs. This has been shown in [49] for VDM, B, and Petri nets.

Compared to other state-based methods, ASM is still young especially when it comes to tool support. VDM originated in 1970's and is one of the longest established formal methods for modeling of computer-based systems. The commercial software, VDM Tools (`www.vdmtools.jp`), offers a rich set of features for design and analysis of VDM specifications, such as type checking, interpretation, debugging, and code generation. Its community-based toolset, Overture (`www.overturetool.org`), is an IDE built on Eclipse that offers syntax and type checking, animation, debugging, proof obligation generation, and test coverage generation. For the Z notation, supporting tools mostly focus on theorem proving (see [50] for a complete list): *ProofPower* (`www.lemma-one.com`), a suite of tools supporting specification and proof in the Z notation, *Z/Eves* [51], a front-end for the *Eves* theorem prover, and *HOL-Z* [50] a proof environment for Z specifications based on the generic theorem prover Isabelle/HOL. A free and open source animator for Z specifications, called *Jaza* [50], is also available for evaluation, testing and (for some specifications) also execution of Z specifications. In addition, the Community Z Tools (CZT) project (`czt.sourceforge.net`) is building a large set of tools supporting the development and analysis of Z specifications, including parsing, typechecking, and animation. Inspired by Z, Alloy is a light-weight specification language aiming at fully automatic analysis of software specifications. It comes with *AlloyAnalyzer*, a model-checker that checks certain properties of specifications by exploring the states of the system and searching for execution instances that satisfy the properties (*simulation*) or counterexamples that violate them (*checking*). The B method [42], the most similar approach to ASMs, is essentially an abstract machine notation with a well-defined notion of refinement that facilitates transformation of abstract models into implementations. It comes with a rich set of both commercial and open source tools (see `www.bmethod.com` for a complete list). *Atelier-B* and the *B-Toolkit* provide syntax analysis, theorem proving, and automatic refinement of B specifications. Model checkers for B, such as *ProB*, offer fully automatic animation of B specifications and support systematic checking of specifications for errors. The Event-B language, an evolution of classical B, offers a simpler notation. The most prominent tool for Event-B, the open

source Rodin platform, builds on the Eclipse platform. Rodin supports refinement and mathematical proofs and it is more mature than CoreASM in certain aspects.

In comparison, CoreASM is still in early stages of maturity—with its community of users and developers being much smaller than of the tools listed above—and offers limited or no support in areas of debugging, typechecking, test generation, and verification. However, the highly extensible architecture of CoreASM offers an extremely flexible platform that facilitates addition of a diverse array of new features and language extensions in the future. The CoreASM language and simulation engine fully support the original ASM constructs with a formally defined semantics that is faithful to the original ASM semantics. In fact, unlike other formal method tools, CoreASM is built upon a formal specification of both its modeling language and simulation engine rigorously defined in ASM terms.

The extensibility features of CoreASM have parallels in the field of Domain Specific Languages (DSL). These are languages whose syntax and semantics are designed to address a specific problem or domain, and are usually less suited to generic programming or specification. Some of these languages can be *hosted* in a generic language, thus providing additional capabilities to the latter, or can *embed* other languages, thus extending their own capabilities [52]. CoreASM differs from both these approaches in two ways. First, in CoreASM the additional syntax and semantics defined via plugins is merged in the language seamlessly (in fact, the language itself is almost entirely defined in this way); there is no *switching* between different modes, languages, or execution contexts as is mostly the case for embedding of DSLs. Second, CoreASM allows extensions of its own computation model via extension points (Section 3.3.2), so plugins can alter basic mechanisms such as loading or preprocessing of specifications, whereas embedded DSLs are usually limited to providing sub-languages for specific fragments of the computation.

Programming languages are sometimes compared to ASM dialects. For instance, a detailed feature-by-feature comparison of Scala vs. AsmL highlights the pros and cons of both languages [53]. This seems appropriate and meaningful as both are strongly-typed languages. In contrast, CoreASM is a modeling language focusing on high-level system specification and design way above the level of strictly typed programming languages and serving very different purposes. Hence, a comparison of CoreASM to such languages is out of the scope of this paper.

7. Conclusions and Future Work

In this paper we have presented the CoreASM toolset, an environment for writing and executing ASM specifications. The core component (the CoreASM interpreter itself) is complemented by a number of different user interfaces (including embedding the tool in Eclipse, running via command line, or even incorporating it in OpenOffice to realize "executable papers"), language extensions (including many experimental constructs that have been proposed in the literature), and export modules (including towards model checking, formal verification, and even typesetting systems).

CoreASM itself is a rather large system with non-trivial extensibility features. Its architecture allows third parties to seamlessly extend the syntax and semantics of the language with domain-specific constructs, thus supporting in executable code the same freedom of expression and of experimentation that has been so successfully exploited in most of the theoretical works on ASMs. In addition, it also allows extensions to the inner working of the interpreter, facilitating the implementation of many practically-relevant features such as monitoring, logging and debugging.

The system has been used in a number of projects, both in scientific and industrial contexts, with good results. The extensibility features of CoreASM were utilized in almost all of these projects, enabling development of new features and integration of the environment with various external tools. Many such extensions were developed to enable the specification writer to focus on the problem at hand, rather than on how to encode the problem in some rigid formal schema. Examples include addition of new data structures and language constructs, linking CoreASM to external visualizers, development of scenario scripting features, and integration of CoreASM specifications with Java programs and libraries.

We have learned a few lessons as well: most importantly, that in building a complex tool a solid theoretical foundation (not disjoint from a sound software engineering sense) is of paramount importance. CoreASM is a well-specified and hence solidly-implemented embodiment of a very rigorous mathematical concept, that of evolving algebra. Starting from a formal specification of the tool environment considerably simplified the design and development process. Such a formal specification also provides a most valuable documentation for future maintenance. While out of the labs the solid foundation might be invisible, and implementation and user interface details may be deemed more important, yet CoreASM would have not survived the test

of time as a formal specification interpreter for a wide array of applications, if it had not been formally specified itself, and systematically implemented according to that specification.

There are a number of open issues that will be the focus of future work with CoreASM. The extensible architecture of CoreASM offers utmost flexibility in extending the language and the engine, however, the current version of CoreASM does not provide any support for conflict detection and resolution between plugins. As mentioned earlier in [17], we consider utilizing the concept of Feature-Oriented Software Development (FOSD) [54], such that every plug-in would provide a list of features. We believe that this would facilitate the integration of complementing features and the detection of inconsistent or overlapping plugins. Currently, there is no support in CoreASM for automatic code generation from ASM specifications, nor for automatic test case generation for conformance testing, comparable to what Spec Explorer [55] offers. The language could be extended to support generic types and more sophisticated mathematical structures, although in such cases a fine balance has to be kept between usability and expressiveness—as is often the case in language design. These extensions together with the development of debugging tools, more powerful editing facilities (e.g., to support navigation in particularly intricate models) and state space visualizers would greatly assist the modeling and design process when dealing with complex control flows.

Acknowledgements. We are grateful to the three anonymous reviewers for their constructive and valuable feedback helping us to improve the final version of this paper. We also thank the editors for the excellent collaboration and their support.

References

[1] D. M. Berry, Formal Methods: the very idea—Some thoughts about why they work when they work, Science of Computer Programming 42 (1) (2002) 11–27.

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

[3] R. Farahbod, U. Glässer, The CoreASM modeling framework, Software: Practice and Experience 41 (2) (2011) 167–178.

[4] R. Farahbod, V. Gervasi, U. Glässer, CoreASM: An Extensible ASM Execution Engine, Fundamenta Informaticae (2007) 71–103.

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

[6] R. Farahbod, U. Glässer, Semantic Blueprints of Discrete Dynamic Systems: Challenges and Needs in Computational Modeling of Complex Behavior., in: New Trends in Parallel and Distributed Computing, Proc. 6th Intl. Heinz Nixdorf Symposium, Jan. 2006, Heinz Nixdorf Institute, 2006, pp. 81–95.

[7] U. Glässer, R. Gotzhein, A. Prinz, The Formal Semantics of SDL-2000: Status and Perspectives, Computer Networks 42 (3) (2003) 343–358.

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

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

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

[11] R. Farahbod, U. Glässer, M. Vajihollahi, An Abstract Machine Architecture for Web Service Based Business Process Management, International Journal of Business Process Integration and Management 1 (2007) 279–291.

[12] E. Börger, Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems, Formal Aspects of Computing 19 (2) (2007) 225–241.

[13] R. Farahbod, CoreASM: An extensible modeling framework & tool environment for high-level design and analysis of distributed systems, Ph.D. thesis, Simon Fraser University, Burnaby, Canada (May 2009).

[14] Y. Gurevich, J. Huggins, The Railroad Crossing Problem: An Experiment with Instantaneous Actions and Immediate Reactions, in: Proceedings of CSL'95 (Computer Science Logic), Vol. 1092 of LNCS, Springer, 1996, pp. 266–290.

[15] A. Blass, Y. Gurevich, Background, Reserve, and Gandy Machines, in: P. Clote, H. Schwichtenberg (Eds.), Computer Science Logic (Proceedings of CSL 2000), Vol. 1862 of LNCS, Springer, 2000, pp. 1–17.

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

[17] R. Farahbod, V. Gervasi, U. Glässer, G. Ma, CoreASM plug-in architecture, in: J.-R. Abrial, U. Glässer (Eds.), Rigorous Methods for Software Construction and Analysis, Springer LNCS Festschrift volume 5115, Springer, 2009, pp. 147–169.

[18] T. A. Standish, Extensibility in programming language design, SIGPLAN Not. 10 (7) (1975) 18–21.

[19] G. Z. Ma, Model Checking Support for CoreASM: Model Checking Distributed Abstract State Machines Using Spin, Master's thesis, Simon Fraser University, Canada (May 2007).

[20] R. Farahbod, U. Glässer, P. Jackson, M. Vajihollahi, High Level Analysis, Design and Validation of Distributed Mobile Systems with CoreASM, in: Proceedings of 3rd International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2008), Springer, 2008.

[21] R. Farahbod, U. Glässer, A. Khalili, A Multi-Layer Network Architecture for Dynamic Resource Configuration & Management of Multiple Mobile Resources in Maritime Surveillance, in: Proc. of SPIE Defense & Security Symposium, Orlando, Florida, USA, 2009.

[22] R. Farahbod, U. Glässer, E. Bossé, A. Guitouni, Integrating Abstract State Machines and Interpreted Systems for Situation Analysis Decision Support Design, in: Proc. of the 11th Intl Conf. on Information Fusion (Fusion 2008), 2008.

[23] P. Maupin, A.-L. Jousselme, Interpreted Systems for Situation Analysis, in: Proc. of the 10th Intl. Conf. on Information Fusion, Quebec city, Canada, 2007.

[24] R. Farahbod, V. Avram, U. Glässer, A. Guitouni, Engineering situation analysis decision support systems (2011).

[25] R. Farahbod, V. Avram, U. Glässer, A. Guitouni, A formal approach to high-level design of situation analysis decision support systems (2011).

[26] P. L. Brantingham, U. Glässer, P. Jackson, M. Vajihollahi, Modeling Criminal Activity in Urban Landscapes, in: N. Memon, J. D. Farley, D. L. Hicks, T. Rosenoørn (Eds.), Mathematical Methods in Counterterrorism, Springer, 2009, pp. 9–31.

[27] M. Altenhofen, E. Börger, Concurrent abstract state machines and +CAL programs, Recent Trends in Algebraic Development Techniques: 19th International Workshop, WADT 2008, Pisa, Italy, June 13-16, 2008, Revised Selected Papers (2009) 1–17.

[28] J. Lemcke, A. Friesen, Composing web-service-like abstract state machines (ASMs), Services, IEEE Congress on (2007) 262–269.

[29] J. Beckers, D. Klünder, S. Kowalewski, B. Schlich, Direct support for model checking abstract state machines by utilizing simulation, in: ABZ '08: Proceedings of the 1st international conference on Abstract State Machines, B and Z, London, UK, 2008, pp. 112–124.

[30] V. Gervasi, D. Mazzei, Using abstract state machines in modeling biological systems, in: R. Burattini, R. Contro, P. Dario, L. Landini (Eds.), Atti del Congresso Nazionale di Bioingegneria 2008, Patron editore, Pisa, Italy, 2008, pp. 79–80.

[31] V. Gervasi, An ASM model of concurrency in a web browser, submitted to the Third International ABZ Conference, Pisa, Italy, 2012.

[32] Y. Gurevich, J. Huggins, Evolving Algebras and Partial Evaluation, in: B. Pehrson, I. Simon (Eds.), IFIP 13th World Computer Congress, Vol. I: Technology/Foundations, Elsevier, Amsterdam, the Netherlands, 1994, pp. 587–592.

[33] B. Beckert, J. Posegga, leanEA: A Lean Evolving Algebra Compiler, in: H. K. Büning (Ed.), *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95)*, Vol. 1092 of LNCS, Springer, 1996, pp. 64–85.

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

[35] G. Del Castillo, I. Durdanović, U. Glässer, An Evolving Algebra Abstract Machine, in: H. K. Büning (Ed.), *Proceedings of the Annual Conference of*

the European Association for Computer Science Logic (CSL'95), Vol. 1092 of LNCS, Springer, 1996, pp. 191–214.

[36] Microsoft FSE Group, The Abstract State Machine Language, available electronically at http://research.microsoft.com/en-us/projects/asml (2009).

[37] M. Anlauff, P. Kutter, eXtensible Abstract State Machines, xASM open source project: http://www.xasm.org.

[38] G. Del Castillo, Towards Comprehensive Tool Support for Abstract State Machines, in: D. Hutter, W. Stephan, P. Traverso, M. Ullmann (Eds.), Applied Formal Methods — FM-Trends 98, Vol. 1641 of LNCS, Springer-Verlag, 1999, pp. 311–325.

[39] J. Schmid, AsmGofer, available electronically at http://www.tydo.de/doktorarbeit/asmgofer.html (2008).

[40] Formal Methods laboratory of University of Milan, Asmeta, available electronically at http://asmeta.sourceforge.net (2006).

[41] D. Jackson, Software Abstractions: Logic, Language, and Analysis, MIT Press, 2006.

[42] J. Abrial, The B-Book: Assigning Programs to Meanings, Cambridge University Press, 1996.

[43] M. Bidoit, P. Mosses, Casl User Manual: Introduction to Using the Common Algebraic Specification Language Casl, SpringerVerlag, 2004.

[44] D. Bjørner, C. B. Jones (Eds.), The Vienna Development Method: The Meta-Language, Vol. 61 of Lecture Notes in Computer Science, Springer, 1978.

[45] J. M. Spivey, The Z Notation: a reference manual, 2nd Edition, Prentice Hall International Series in Computer Science, 1992.

[46] N. Wirth, Program development by stepwise refinement, Communications of the ACM 14 (4) (1971) 221–227.

[47] E. W. Dijkstra, Notes on structured programming, in: Structured Programming, Academic Press, London, 1972, Ch. 1, pp. 1–82.

[48] E. Börger, The ASM method: An exposition, in: P. Boca, et al. (Eds.), Formal Methods: State of the Art and New Directions, Springer, 2010.

[49] E. Börger, High Level System Design and Analysis using Abstract State Machines, in: D. Hutter and W. Stephan and P. Traverso and M. Ullmann (Ed.), Current Trends in Applied Formal Methods (FM-Trends 98), no. 1641 in LNCS, Springer-Verlag, 1999, pp. 1–43.

[50] Z notation tool support, http://formalmethods.wikia.com/wiki/Z_notation [June 2010].

[51] O. Canada, Z/eves version 1.5: An overview, in: FM-Trends, 1998, pp. 367–376.

[52] P. Hudak, Building domain-specific embedded languages, ACM Computing Surveys 28.

[53] S. Micheloud, Scala and asml side by side, www.scala-lang.org/docu/files/ScalaAsmL.pdf (2003).

[54] D. Batory, J. Sarvela, A. Rauschmayer, Scaling stepwise refinement, 2003. URL citeseer.ist.psu.edu/batory03scaling.html

[55] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, L. Nachmanson, Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer, in: R. M. Hierons, J. P. Bowen, M. Harman (Eds.), Formal Methods and Testing, Vol. 4949 of Lecture Notes in Computer Science, Springer, 2008, pp. 39–76.