Design Exploration and Experimental Validation of Abstract Requirements

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

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

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

Abstract. How can one cope with the notorious problem of establishing the correctness and completeness of abstract functional requirements in the design of control-intensive software systems prior to actually building the system? The answer given here explores the *abstract state machine* (ASM) paradigm: a versatile semantic framework for computational modeling of virtually all kinds of discrete dynamic systems. Combining common abstraction principles from computational logic and discrete mathematics, abstract state machines provide a universal model of computation and an effective instrument for analyzing and reasoning about complex semantic properties of real-world systems. In this paper we introduce a novel ASM tool environment for making ASM ground models executable on real machines. We present the CoreASM language and the general architecture of the CoreASM engine together with a high-level description of its extensibility mechanisms.

1 Introduction

Abstraction and formalization provide effective instruments for establishing critical system requirements by modeling the construction of software designs prior to coding so that one can analyze and reason about specification and design choices, fully understanding their deeper implications [1]. To this end, computational logic and discrete mathematics help eliminating typical deficiencies — such as ambiguities, loose ends and inconsistencies — that often remain hidden in informal requirements. Mathematical precision typically helps gaining a clearer and more thorough understanding of the problem to be solved, thus reducing the risk of making premature decisions with fatal consequences [2]. A higher level of precision is needed with the increasingly intricate nature of system requirements.

A higher level of precision often brings with it increased complexity. Hence, a question comes naturally: How can one cope with the notorious problem of

establishing the correctness and completeness of (abstract, functional) requirements in the design of a software-intensive systems prior to actually building the system? The answer given here explores the *abstract state machine* (ASM) paradigm [3], a universal mathematical framework for computational modeling of functional system requirements in terms of abstract operational semantic properties. In addition to the semantic issues involved in gaining mathematical precision as needed, we address and exemplify here the use of ASM abstraction principles as an effective means for gradually developing a clear architectural model of the system under design. The ASM approach encourages viewing the behavior of a software system as evolution of *abstract states*, as represented by the runs of an ASM model. This formalized yet abstract view is invaluable for bridging the gap between informal requirements and precise specifications in the earlier phases of system design. This angle also simplifies the task of constructing models of requirements that are being extracted from implementations in reverse engineering applications.

Abstract state machines are known for their versatility in modeling of algorithms, architectures, languages, protocols and virtually all kinds of sequential, parallel and distributed systems. The ASM formalism has been studied extensively by researchers in academia and industry for more than 15 years with the intention to bridge the gap between formal and pragmatic approaches. Leaning toward practical applications of formal methods, this work resulted in a solid methodological foundation for building ASM models. Widely recognized applications include semantic foundations of industrial system design languages, like the ITU-T standard for SDL, the IEEE language VHDL and its successor SystemC, programming languages like JAVA, C# and Prolog, Web service description languages, communication architectures, embedded control systems, et cetera (see the ASM website at `www.eecs.umich.edu/gasm/` and the annotated bibliography in [3]). Beyond the natural ASM habitat of hardware/software systems, computational modeling of social systems and cognitive behavior [4, 5] addresses the broader scope of discrete dynamic systems that the very original definition of ASMs intended to capture [6].

Model-based systems engineering can benefit from *abstract executable specifications* as a tool for design exploration and experimental validation through simulation and testing (see Section 3.1). Building on experiences with two generations of ASM tools, a novel executable ASM language, called CoreASM, is being developed [7, 8]. The CoreASM language emphasizes freedom of experimentation and supports the evolutionary nature of design as a product of creativity. It is particularly suited to Exploring the problem space for the purpose of writing an *initial specification*. The CoreASM language allows writing of highly abstract and concise specifications by minimizing the need for encoding in mapping the problem space to a formal model, and by allowing explicit declaration of the parts of the specification that are purposely left abstract. The principle of minimality, in combination with robustness of the underlying mathematical framework, improves modifiability of specifications, while effectively supporting the highly iterative nature of specification and design.

This paper describes the open-source CoreASM environment, which consists of a platform-independent *engine* for executing specifications 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, thereby enabling development and integration of complementary tools, e.g., for symbolic model checking [9] and automated test generation [10]. The design of CoreASM is novel and the underlying principles are unprecedented among the existing executable ASM languages, including the most advanced ones: AsmL [11], the ASM Workbench [12], XASM [13], and AsmGofer [14].

This paper is structured as follows. Section 2 summarizes the asynchronous ASM computation model and its support for concurrency, reactive behavior, and real-time aspects. Section 3 presents the CoreASM language, its features, and its tool environment. Section 4 explores related work and Section 5 concludes the paper.

2 Abstract State Machines

A central problem in computing science and discrete mathematics is the question of how to precisely define the notion of *algorithm*. Traditionally, Turing machines (TMs) have been used in the study of the theory of computation as a formal model of algorithm [15]. For the study of semantic issues, however, this model of computation is totally inappropriate, as there is typically a huge gap between the abstraction level of a problem and the low-level computation of a TM simulating the corresponding algorithm. In contrast, any algorithm, regardless of how abstract, can be modeled by an ASM at its natural level of abstraction.

Theoretical foundations of ASM computation models show that *sequential ASMs* capture the notion of sequential algorithms in the aforementioned sense [16, 17]. Likewise, *parallel ASMs* (a.k.a. the basic ASM model) capture parallel algorithms [18]. For the asynchronous computation model of *distributed ASMs*, a generalization of the two other models, a formal proof that this model captures distributed algorithms is not known. However, there is considerable empirical evidence from practical applications that the asynchronous ASM model faithfully reflects the intuitive understanding of the notion of distributed algorithm. We briefly summarizes here the formal semantic framework of ASMs using common notions and structures from discrete mathematics and computing science. For further details, we refer to [3, 6].

The asynchronous computation model of distributed abstract state machines (DASM), originally proposed in [6] (where ASMs are called *evolving algebras*), defines concurrent and reactive behavior, as observable in distributed computations performed by any finite number of autonomously operating computational agents, in terms of *partially ordered runs*.

A DASM M is defined over a given vocabulary V by its program P_M and a non-empty set I_M of initial states. V consists of a finite collection of symbols denoting mathematical objects and their relation in the formal representation of M, where we distinguish *domain symbols*, *function symbols* and *predicate*

symbols. Symbols that have a fixed interpretation regardless of the state of M are called *static*; those that may have different interpretations in different states of M are called *dynamic*. A state S of M results from a valid interpretation of all the symbols in V and constitutes a variant of a first-order structure, one in which all relations are formally represented as Boolean-valued functions.

Concurrent control threads in an execution of P_M are modeled by a dynamic set AGENT of autonomously operating computational *agents*. This set may change dynamically over runs of M as required to deal with varying computational resources. Agents of M interact with one another, and possibly also with the operational environment of M, by reading and writing shared locations of a global machine state. The underlying semantic model regulates such interactions so that potential conflicts are resolved according to the definition of partially ordered runs.

P_M consists of a statically defined collection of agent programs $P_{M_1}, ..., P_{M_k}$, $k \geq 1$, each of which defines the behavior of a certain *type* of agent in terms of state transition rules. The canonical rule consists of a basic update instruction of the form

$$f(t_1, t_2, ..., t_n) := t_0,$$

where f is an n-ary dynamic function symbol and the $t_i{'}s$ $(0 \leq i \leq n)$ are terms. An update instruction specifies a pointwise function update, i.e., an operation that replaces an existing function value by a new value to be associated with the given function arguments. Complex rules are inductively defined by a number of simple rule constructors allowing the composition of rules in various ways.

A computation of an individual agent of M, executing program P_{M_j}, is modeled by a finite or infinite sequence of state transitions of the form

$$S_0 \overset{\Delta_{S_0}(P_{M_j})}{\longrightarrow} S_1 \overset{\Delta_{S_1}(P_{M_j})}{\longrightarrow} S_2 \overset{\Delta_{S_2}(P_{M_j})}{\longrightarrow} \cdots,$$

such that S_{i+1} is obtained from S_i, for $i \geq 0$, by firing $\Delta_{S_i}(P_{M_j})$ on S_i, where $\Delta_{S_i}(P_{M_j})$ denotes a finite set of updates computed by evaluating P_{M_j} over S_i. Firing an update set means that all the updates in this set are fired simultaneously in one atomic step.

Finally, M models interactions with a given environment through actions and events that are observable at external interfaces, formally represented by externally controlled functions. Of particular interest are *monitored functions*. Such functions change their values dynamically over runs of M, although they cannot be updated internally by agents of M. A typical example is the abstract representation of global time. In a given state S of M, the global time (as measured by some external clock) is given by a monitored unary function *now* taking values in a linearly ordered domain TIME. Values of *now* increase monotonically over runs of M.

3 The CoreASM Project

The natural computation model and ease of abstraction of ASM is best used in the early phases of requirements modeling, as shown by the many case reports (see for example [19] where a number of RE techniques, including ASMs, are evaluated comparatively on a benchmark problem). Yet, other ASM execution environments target mostly the detailed design to pre-coding stages.

The CoreASM language and supporting tool architecture focus on early phases of the software design process. In particular, we want to encourage rapid prototyping with ASMs, starting with mathematically-oriented, abstract and untyped models and gradually refining them down to more concrete versions — a powerful technique for specification with refinement that has been exploited in [3] and [20]. CoreASM maintains executability of even fairly abstract and incomplete models, which in turn is important to make animation of the specification possible, and to improve communication with the stakeholders during the requirements elicitation and analysis process. CoreASM is thus a *lightweight formal method*, which can be fruitfully applied since the requirements elicitation phase without having to pay huge time and effort penalties.

As we mentioned above, the CoreASM project is not the first attempt to make ASMs executable. However the main commonality between our approach and others is simply that they provide platforms which execute specifications with ASM-like semantics. Beyond this, CoreASM departs in its approach and spirit of principle from its predecessors. The project's core ideology, consisting of two simple tenets, both serves as a motivation and as a guide development of the engine, the language, and the supporting tool environment:

- the preservation of pure ASM semantics, and
- ensuring freedom through extensibility.

First and foremost, the CoreASM engine should as closely as possible resemble the pure mathematical definition of ASMs. Where other languages impose strict typing conventions on the language, we do not sacrifice the freedom of typing inherent in ASMs. Where other languages have adopted an object oriented view of ASM elements, we preserve the view imparted by the original framework.

Since their inception, ASMs have been extended, within the bounds of their original semantics [6], with many useful additions including those allowing for the modeling of distributed systems and the modeling of distributed incremental modification of data structures. Such extensibility coupled with the fact that ASMs allow for the definition of new rule forms, data types and operators give the system architect the freedom and flexibility required to fully explore the problem space without restriction. CoreASM has been designed to offer the same level of flexibility through extensibility via a well-defined plug-in architecture, thereby preserving the freedom of experimentation that has proven to be so fruitful in the development of ASM concepts. Other languages support limited extensibility by writing classes or interacting with external C code, but none of the previous proposals supports pervasive extensibility of syntax and semantics as CoreASM does.

Our approach has also taken into account the extensibility of other software with the CoreASM engine as a component. The platform independent engine, with a well defined interface offering full control and observability of an ASM run, lends itself to the possibility of interaction with many other useful analysis- and design-support tools.

In this section we discuss the two most important benefits which stem from the CoreASM project goals. We will see why executable specifications are important for requirements engineering, and discuss how our extensible language and engine along with supporting tools, provide real value to the systems architect.

3.1 Executable Specifications

Executable specifications offer many advantages that are of great relevance in software engineering, and have been the subject of a continuous stream of research. We will outline here only the major benefits for requirements specification:[3]

- It is possible to *animate* an executable specification (e.g., execute it while monitoring its execution), and observe its behavior, including the behavior specified for unexpected cases which may be forced by explicitly altering the environment.
- Execution traces constitute scenarios which can be presented and discussed with the stakeholders, helping in requirements verification (by comparing the specified behavior with the customers' desired behavior), elicitation (as the observation of the current behavior may prompt the discovery of new requirements) and negotiation (as it is easier to re-assess different stakeholders' priorities in case of conflicts when the discussion concentrate on a concrete case).
- Executable specifications are also amenable to "debugging", i.e., it becomes possible to identify mismatches between the specifier's intended behavior and the specified one. Notice that this is different from the point above (where the customers' intended behavior and the specified one are compared), and covers cases where the analyst or specifier has introduced errors while *encoding* the specification.
- Executable specifications have by definition a single, sound semantics — which is the one enforced by the execution engine. While the semantics itself may contain abstract elements (as is the case for ASMs), the scope for abstraction is clearly defined, and there is no confusion possible between abstraction and ambiguity, which is instead a weak point of purely descriptive specifications.
- Finally, executability facilitates interoperability with other important techniques, e.g. model checking, interaction tests through GUI mock-ups, automated regression testing, etc. All these techniques have been shown in a

[3] See the classical reference [21] for a more extensive overview.

number of studies to be highly effective when appropriate, and a fully formal, executable specification constitute a valuable asset to capitalize on in these cases.

Despite all these advantages, executable specifications have not enjoyed major popularity, mostly because they are typically harder to write than informal or semi-formal specifications (e.g., natural language, UML use cases, scenarios, etc.). In the CoreASM approach, great importance is placed on the return on investment (ROI) that the analyst can expect. While the ROI has to be evaluated on a case-by-case basis, many classes of practically relevant systems would clearly benefit from the adoption of stronger specification methods. These include safety-critical systems and high-assurance systems, but also more mundane applications where interoperability is important (e.g. protocol specifications, web services descriptions, etc.).

The CoreASM approach places scalability of investment among its main goals. With CoreASM, it is possible to write *lightweight* formal specifications: for example, it supports untyped specifications, abstract and oracle functions, etc. At the same time, it is possible to write *heavy-duty*, completely defined specifications — a usage which borders with programming languages. The former approach is more suited to upper requirements engineering, while the latter is more suited to detailed specification. This is an innovative approach compared to other ASM execution engines (e.g. [11]) which have offered only the fully formal level.

We claim that the CoreASM approach, with its lightweight model, extensibility with domain-specific constructs (which will be discussed in the next section), and rich support environment, changes the traditional economics for executable specifications. In effect, CoreASM extends many of the advantages of executable specifications cited above to a whole new class of systems for which the investment of effort required by previous approaches was not justified. In this sense, we are extending the range of systems for which writing executable specifications is economically sensible.

3.2 Extensible Language

In keeping with the micro-kernel spirit of the CoreASM approach, most of the functionality of the engine is implemented through plug-ins to a minimal kernel. Such an architecture supports future extensions of the language through various (possibly third-party) plug-ins. The architecture supports three main 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[4], 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

[4] We call *background* a collection of mathematical domains and relations, packaged together as a single unit. This concept is akin to that of multi-sorted algebra.

encoding and decoding functions for representing elements of the background for storage purposes, and (iii) an extension to the interpreter providing the semantics for all the operations defined in the background.
- Rule plug-ins are used to implement specific rule forms, with the understanding that the execution of a rule always results in a (possibly empty) set of updates to the state. 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. It is worthwhile to note that only a single scheduling policy can be in force at any given time, whereas an arbitrary number of background and rule plug-ins can be all in use at the same time.

In CoreASM, the kernel only contains the bare essentials, that is, all that is needed to execute only the most basic ASM. Other than a few essential rules (e.g., assignment), all other rule forms (e.g., **if/then/else**, **choose**, **forall**) are implemented as plug-ins in a standard library, which is implicitly loaded with each CoreASM specification. Only the backgrounds of *booleans* (needed to express characteristic functions) and *rules* (needed to represent the rules in an ASM specification) are included in the kernel. It should be noted however that the kernel does not include all of the expected corresponding operations. For example, while the domain of booleans (that is, true and false) is in the kernel, boolean algebra ($\wedge$, $\vee$, $\neg$, etc.) is not, and is instead provided through a background plug-in. In the same vein, while universes are represented in the kernel through set characteristic functions, the background of finite sets is implemented as a plug-in, which provides expression syntax for defining them, as well as an implicit representation for storing sets in the abstract state, and implementations of the various set theoretic operations (e.g., $\in$) that work on such implicit representation.

Finally, there is a single scheduling policy implemented in the kernel, namely the pseudo-random selection of an arbitrary set of agents at a time, which is sufficient for multi-agent ASMs where no assumptions are made on the scheduling policy.

In addition to modular extensions of the engine, plug-ins can also register themselves for *Extension Points*. Each mode transition in the execution engine is associated to an extension point. At any extension point, if there is any plug-in registered for that point, the rule provided by the plug-in at registration time is executed before the engine proceeds into the new mode. Such a mechanism enables extensions to the engine's life-cycle which facilitates implementing various practically relevant features such as adding debugging support, adding a C-like preprocessor, or performing statistical analysis of the behavior of the simulated machine (e.g., coverage analysis or profiling). A plug-in, for example, could monitor the updates that are generated by a step before they are actually applied to the current state of the simulated machine, possibly checking conditions on

these updates and thus implementing a watch (i.e., displaying updates to certain locations) or a watch-point (i.e., suspending execution of the engine when certain updates are generated), which are both useful for debugging purposes.

As already mentioned, the CoreASM engine is accompanied by a *standard library* of plug-ins including the most common backgrounds and rule forms (i.e., those defined in [3]), an extension library including a small number of specialized backgrounds and rules, and API specifications for writing new plug-ins that can easily be integrated in the environment. Extension plug-ins must be explicitly imported into an ASM specification by an explicit **use** directive.

3.3 Tool Environment

To facilitate the integration of executable ASMs with other analysis- and design-support tools, the CoreASM engine provides an API offering full control over the ASM being executed and its current run. This allows tool integrators to use CoreASM as a component of a larger requirements modeling and validation environment.

Using this API, an ASM can be explored by stepping forward or rolling back the current run. The API also allows for the monitoring and modification of abstract states between steps. This facility to actively interact and modify abstract states gives external tools the ability to behave as if they are part of the environment of the machine being monitored.

A GUI-based *integrated modeling environment* (IME) is currently under development [22], at the heart of which lies the CoreASM engine (see Figure 1). This user-friendly visual solution for model design and validation will provide a complete platform for accessing and using a variety of complementary tools. The interface organizes information relevant to state transition into different views, visually highlights inconsistencies of the model, and gives the user the ability to compare and contrast state and updates produced by different steps (e.g., see the state view at the top right corner of Figure 1). In addition to the ease that the IME shall bring to manual validation, integrated machine-aided tools such as model checking and run-time assertion checking for abstract states of the ASM run are also slated for development. Work is currently ongoing on the addition of user interaction commodities as automatic syntax highlighting, automatic completion, and visual control flow-based model development using *control state ASM* activity diagrams.

In Figure 2(a) we show as an example a fragment of a CoreASM specification for an automated teller machine (ATM).[5] The example assumes an asynchronous interaction model between three autonomously operating entities involved in ATM transactions, namely: the ATM, a user, and the bank (see Figure 2(b)). Due to space considerations, the example is restricted to withdrawal transactions only.

Basically, the ATM control forms a distributed embedded system which is modeled in terms of a DASM consisting of three separate agents (each of which

[5] This example was originally introduced in [23].

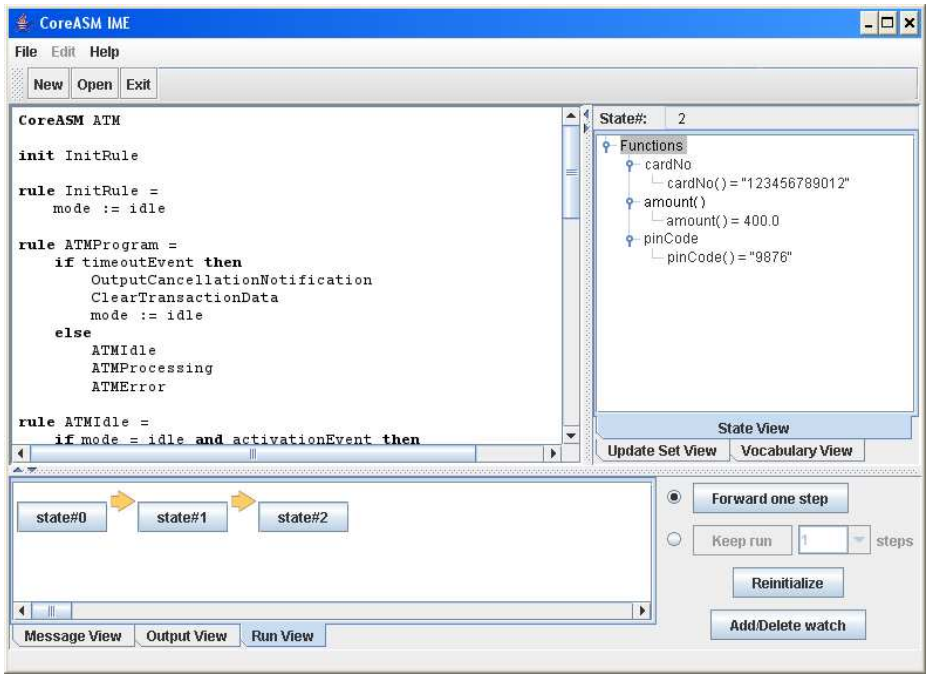

Fig. 1. Snapshot of the CoreASM IME (under development)

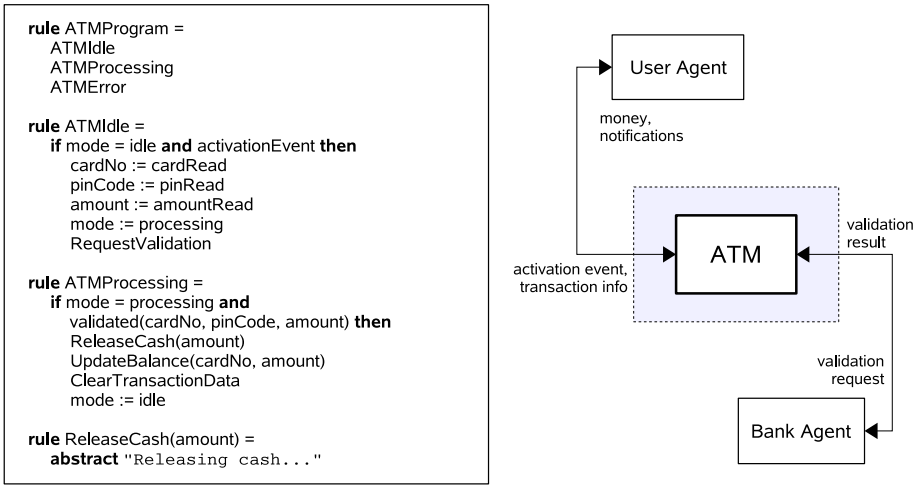

Fig. 2. a) Part of an ATM specification in CoreASM; b) ATM agent and its environment

represents one of the interacting entities). Nonetheless, one may start by modeling only the ATM agent, and by assuming that the other agents are part of the global environment of the ATM manager, as per Figure 2(b). This view allows us to focus on the key behavioral aspects first. In subsequent refinement steps, this model can easily be extended to a DASM by making the behavior of the two other agents explicit (i.e., by adding the specification of the corresponding ASM agents).

The familiar syntax (akin to that of imperative programming languages) results in a specification that is both simple to create for requirements engineers and system designers and difficult to misinterpret for those responsible for implementation. Notice how CoreASM provides support for abstraction by allowing the specifier to omit the body of a rule and mark it as *abstract* instead, by using the syntax

$$\textbf{rule } R = \textbf{abstract } \textit{narrative text}.$$

When the abstract rule is invoked, the engine executes it symbolically, which in the case of interactive execution is obtained by presenting to the user the given narrative, and optionally offering to manually update the state as it is expected that the concrete rule should do had it already be defined.

In contrast, many heavy-duty specification methods would have request a fully-concrete, complete specification of the ATM and of its environment before being able to perform any behavioral analysis.

4 Related Work

Machine assistance plays an increasingly important role in making writing and analyzing complex specifications practical. Model-based systems engineering demands for abstract executable specifications as a basis for design exploration and experimental validation through simulation and testing. The role of executable requirements has been recognized since the emergence of requirements engineering as an autonomous discipline [24], and different forms of executable requirements are still today advocated as a most effective means to ease rapid prototyping, HCI design, validation and verification [25–27]. Thus it is not surprising that there is a considerable variety of executable requirements languages that have been developed over the years.

Based on experience with various experimental ASM interpreters [28–31], and fundamental concepts of making ASMs executable on real machines [32], a second generation of more mature ASM tools and tool environments was developed: *AsmL* (ASM Language) [11], the *ASM Workbench* [12] and the *Xasm* (Extensible ASM) language [13] are all based on compilers, while *AsmGofer* [14] provides an ASM interpreter.[6] The most prominent one is AsmL, developed by the Foundations of Software Engineering group at Microsoft Research. AsmL

[6] We focus here on the more common and well-known ASM tools. For a complete overview, see also [3], Sect. 8.3.

is a strongly typed language based on the concepts of ASMs but also incorporates numerous object-oriented features and constructs for rapid prototyping of component-oriented software, thus departing in that respect from the theoretical model of ASMs; rather it comes with the richness of a fully fledged programming language. At the same time, it lacks any built-in support for dealing with distributed systems. Being deeply integrated with the software development, documentation, and runtime environments of Microsoft, its design was shaped by practical needs of dealing with fairly complex requirements and design specifications for the purpose of software testing; as such, it is oriented toward the world of code. This has made it less suitable for initial modeling at the peak of the problem space and also restricts the freedom of experimentation.

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

5 Conclusion and Future Work

We have outlined in this paper the design principles of CoreASM, a practical tool environment for building ASM ground models for the purpose of separating specification and design (problem solving) from construction and coding (engineering). The development of our extensible specification execution environment will be beneficial for both researchers who will be able to test future extensions to ASMs with our engine, and for industry who will use the engine and its extensions to make more precise and reliable software specifications. Sensible instruments and tools for writing an initial specification call for maximal flexibility and minimal encoding, as a prerequisite for evolutionary modeling which requires easily modifiable formal specifications. The aim of the CoreASM effort is to address this need for abstractly executable specifications.

CoreASM addresses practical needs of hardware/software system designers, architects, and requirement engineers dealing with abstract behavioral descriptions of functional requirements. An often preferred style for describing behavior in early design stages is pseudocode. Designing with pseudocode enhances intellectual control over complex computational problems as it avoids getting caught in irrelevant details by hiding such details in abstract data structures. To this end, CoreASM programs resemble pseudocode but, at the same time, have a precisely defined meaning, so one can check and experimentally validate requirement specifications by executing CoreASM programs. Combining declarative, functional and imperative styles as needed provides enormous flexibility for choosing a level of abstraction that is most appropriate.

To the requirements practitioner, CoreASM offers a way to express precisely the intended semantics, including notation to declare what has been purpose-

fully left abstract. Coupled with sound refinement principles and interoperability with other tools, CoreASM supports rational elicitation, animation, validation and verification. To the domain expert, CoreASM provides a workbench to define domain-specific languages by writing simple plug-ins which specify merely the extensions to the basic ASM semantics, thus vastly reducing the entry barrier to the definition and development of domain-specific constructs. To the RE theorist, CoreASM proposes the theoretically well-founded, operational ASM specification method whose computational properties and capabilities have been formally proved, and has a record of having successfully supported the specification of literally hundreds of problems of all classes (embedded systems, communication protocols, computer languages, distributed systems, web services, social systems, etc.).

In future work we intend to extend the CoreASM engine by providing various rule and background plug-ins to support commonly used abstract data structures and mathematical functions. We are also planning to enrich the CoreASM tool suite by providing more assistance to the specification writer through syntax-guided editing, completion and highlighting, and by facilitating the interoperability with other tools. Development work is planned in the area of trace visualization, and implementation of the CoreASM engine as a plug-in for the open source Eclipse integrated development environment [33] is being investigated.

Acknowledgment. The authors would like to thank the anonymous reviewers for their many suggestions which led to a significant improvement of the paper.

References

1. Berry, D.: Formal Methods: The Very Idea. Science of Computer Programming **42**(1) (2002) 511–27
2. Huckle, T.: Collection of software bugs. Technical report, Technical University Munich (2004) Last visited Sep. 2005, `http://www5.in.tum.de/~huckle/bugse.html`.
3. Börger, E., Stärk, R.: Abstract State Machines: A Method for High-Level System Design and Analysis. Springer-Verlag (2003)
4. Brantingham, P.L., Glässer, U., Kinney, B., Singh, K., Vajihollahi, M.: A Computational Model for Simulating Spatial Aspects of Crime in Urban Environments. In: Proc. IEEE International Conference on Systems, Man and Cybernetics. IEEE Press (2005) 3667–3674
5. Glässer, U., Rastkar, S., Vajihollahi, M.: Computational Modeling and Experimental Validation of Aviation Security Procedures. In: To Appear in the Proceedings of the IEEE International Conference on Intelligence and Security Informatics (ISI-2006). (2006)
6. Gurevich, Y.: Evolving Algebras 1993: Lipari Guide. In Börger, E., ed.: Specification and Validation Methods. Oxford University Press (1995) 9–36
7. Farahbod, R., Gervasi, V., Glässer, U.: CoreASM: An extensible ASM execution engine. In Beauquier, D., Börger, E., Slissenko, A., eds.: Proc. of the 12th Int'l Workshop on Abstract State Machines. (2005) 153–165

8. Farahbod, R., et al.: The CoreASM Project. (2005) http://www.coreasm.org.
9. Del Castillo, G., Winter, K.: Model Checking Support for the ASM High-Level Language. In Graf, S., Schwartzbach, M., eds.: Proceedings of the 6th International Conference TACAS 2000. Volume 1785 of LNCS., Springer-Verlag (2000) 331–346
10. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using Spin to generate tests from ASM specifications. In: Abstract State Machines 2003, Springer (2003) 263–277
11. Microsoft FSE Group: The Abstract State Machine Language. (2003) Last visited June 2003, http://research.microsoft.com/fse/asml/.
12. Del Castillo, G.: Towards Comprehensive Tool Support for Abstract State Machines. In Hutter, D., Stephan, W., Traverso, P., Ullmann, M., eds.: Applied Formal Methods — FM-Trends 98. Volume 1641 of LNCS., Springer-Verlag (1999) 311–325
13. Anlauff, M.: XASM – An Extensible, Component-Based Abstract State Machines Language. In Y. Gurevich and P. Kutter and M. Odersky and L. Thiele, ed.: Abstract State Machines: Theory and Applications. Volume 1912 of LNCS., Springer-Verlag (2000) 69–90
14. Schmid, J.: Executing ASM Speciticications with AsmGofer. (2005) Last visited Sep. 2005, www.tydo.de/AsmGofer/.
15. Sipser, M.: Introduction to the Theory of Computation. PWS Publishing Company (1997)
16. Gurevich, Y.: Sequential Abstract State Machines Capture Sequential Algorithms. ACM Transactions on Computational Logic **1**(1) (2000) 77–111
17. Reisig, W.: On Gurevich's Theorem on Sequential Algorithms. Acta Informatica **39**(5) (2003) 273–305
18. Blass, A., Gurevich, Y.: Abstract State Machines Capture Parallel Algorithms. ACM Transactions on Computation Logic **4**(4) (2003) 578–651
19. Brger, E., Gotzhein, R.: The light control case study: A synopsis. Journal of Universal Computer Science **6**(7) (2000) 582–585
20. Börger, E.: The ASM Refinement Method. Formal Aspects of Computing (2003) 237–257
21. Fuchs, N.E.: Specifications are (preferably) executable. Software Engineering Journal **7**(5) (1992) 323–334
22. Su, M.: Use abstract state machines to model a graphical user interface system. Master's thesis, Simon Fraser University, Burnaby, Canada (2006)
23. Glässer, U., Vajihollahi, M.: Engineering Concurrent and Reactive Systems with Distributed Real-Time Abstract State Machines. In: DIPES 2004: IFIP Working Conference on Distributed and Parallel Embedded Systems, Kluwer (2004)
24. Zave, P.: An operational approach to requirements specifications for embedded systems. IEEE Transactions on Software Engineering **8**(3) (1982) 250–269
25. Heitmeyer, C., Archer, M., Bharadwaj, R., Jeffords, R.: Tools for construction requirements specifications: The SCR toolset at the age of ten. Computer Systems Science & Engineering **20**(1) (2005) 19–35
26. Tran Van, H., van Lamsweerde, A., Massonet, P., Ponsard, C.: Goal-oriented requirements animation. In: Proceedings of the 12th International Conference on Requirements Engineering, Kyoto, Japan, IEEE CS Press (2004) 218–228
27. Uchitel, S., Chatley, R., Kramer, J., Magee, J.: Fluent-based animation: Exploring the relation between goals and scenarios for requirements validation. In: Proceedings of the 12th International Conference on Requirements Engineering, Kyoto, Japan, IEEE CS Press (2004) 208–217

28. Kappel, A.M.: Executable Specifications Based on Dynamic Algebras. In Voronkov, A., ed.: Logic Programming and Automated Reasoning. Volume 698 of Lecture Notes in Artificial Intelligence. Springer (1993) 229–240
29. Huggins, J.: An offline partial evaluator for evolving algebras. Technical Report CSE-TR-229-95, University of Michigan (1995)
30. Diesen, D.: Specifying Algorithms Using Evolving Algebra. Implementation of Functional Programming Languages. Dr. scient. degree thesis, Dept. of Informatics, University of Oslo, Norway (1995)
31. Beckert, B., Posegga, J.: leanEA: A Lean Evolving Algebra Compiler. In Büning, H.K., ed.: *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95)*. Volume 1092 of LNCS., Springer (1996) 64–85
32. Del Castillo, G., Durdanović, I., Glässer, U.: An Evolving Algebra Abstract Machine. In Büning, H.K., ed.: *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'95)*. Volume 1092 of LNCS., Springer (1996) 191–214
33. Eclipse Foundation: Eclipse.org home page (2006) Last visited Mar. 2006, `http://www.eclipse.org`.