Theory and Application of Abstract State Machines

A. Blass (Univ. of Michigan, Ann Arbor MI, USA)
E. Börger (Univ. di Pisa, Italy)
Y. Gurevich (Microsoft Research, Redmond WA, USA)

3rd-8th March, 2002

This volume contains the abstracts of presentations at the seminar "Theory and Applications of Abstract State Machines," held at Schloss Dagstuhl from 3 to 8 March, 2002.

The seminar was proposed to the participants with the following goal which we restate here from the Call for Participation.

The advances in the theory, the tool development, and the progressive industrial employment of Abstract State Machines (ASMs) in the 90's have turned ASMs into a practical technique for disciplined rigorous software engineering in the large. The proposed seminar aims at bringing together ASM researchers from academia and industrial users of ASMs to strengthen this fruitful interaction between theory and practice.

As a result of the research and the applications of Abstract State Machines during the last decade, ASMs offer a certain number of theoretically well founded and industrially useful methods, which support the entire software development cycle. These include rigorous modeling, analysis and validation methods a) for the requirements, during the early phases of industrial software development, and b) for the refinement of the high level models through a design process which reliably connects the requirements to the code development. Via the definition of appropriate ground models, which can be made executable, ASMs support the elicitation, specification, inspection and testing of requirements. Building the high-level models leads to a good understanding of the requirements. It contributes to practical inspections and to testing which help to detect errors at the earliest possible stage of software production - well known to be responsible for most of the costly errors occurring during the software development process. The controlled stepwise refinement of high-level models which turns them into efficiently executable code also supports a good documentation discipline, which is helpful for the maintenance and the reusability of the intermediate models which reflect critical design decisions.

The specific goal of the seminar is to survey and to critically evaluate the current academic and industrial developments and new results concerning ASMs. In particular we want to provide guidelines for future research and development by identifying new challenges coming e.g. from component based design techniques, software architecture patterns, mobile computing, security concerns, etc. Corresponding to the goal to evaluate ASM related scientific achievements and their current industrial employment and development, the list of persons to be invited tries to reflect both the academic and industrial aspects of current work on ASMs.

The seminar realized those goals. It was attended by over 60 participants from all over Europe and the US. The presentations ranged from highly theoretical work to genuine industrial applications, and so did the discussions.

On behalf of all the participants, we thank the administration of Schloss Dagstuhl for accommodating an unusually large number of attendees and providing pleasant conditions conducive to discussions and research. We also thank the European Community for providing funding for student participants.

The organizers use this opportunity to thank all the participants for their contributions to the success of the workshop.

Andreas Blass
Egon Börger
Yuri Gurevich

Xasm - An Open Source ASM Implementation

Matthias Anlauff, Kestrel Insitute, Palo Alto, CA , USA
March 7, 2002

In this talk, I will present Xasm, an Open Source implementation of Abstract State Machines. The talk will also present the new object-oriented features of Xasm, and how they fit into the general semantical framework of the basic ASM ideas. During the talk, the support system for Xasm and Montages, Gem-Mex, will also be presented. As a newly implemented feature, the Xasm-Java language interface will be demonstrated. The Gem-Mex tool is available as Open Source software at `www.xasm.org`.

Runtime Verification using ASMs

Mike Barnett
Microsoft Research
March 6, 2002

We propose a method for implementing behavioral interface specifications on the the .NET Platform. Our interface specifications are expressed as executable model programs written in AsmL, a specification language based on the theory of Abstract State Machines (ASMs). Model programs can be run either as stand-alone simulations or used as contracts to check the conformance of an implementation class to its specification. We focus on the latter, which we call *runtime verification*. Our method allows for non-deterministic specifications as well as for the specification of interaction patterns.

Knowledge Discovery, Maximum Entropy and the Inverse Representation Problem

Christoph Beierle, FernUniversität Hagen, Germany
March 5, 2002

Commonsense and expert knowledge is most generally expressed by rules, connecting a precondition and a conclusion by an *if-then*-construction. If-then-rules are more formally denoted as conditionals, and often they occur in the form of probabilistic conditionals like "*Students are young with a probability of (about) 80 %*" and "*Singles (i.e. unmarried people) are young with a probability of (about) 70 %*", where this commonsense knowledge can be expressed formally by $\{(young|student)[0.8], (young|single)[0.7]\}$. The crucial point with conditionals is that they carry generic knowledge which can be applied to different situations. This makes them most interesting objects in Artificial Intelligence, in theoretical as well as in practical respect.

Whereas probabilistic conditional reasoning is quite weak if all models are taken into account, the reasoning facilities can be improved by the well-known principle of maximum entropy. This principle selects a model that completes the (incomplete) knowledge given by a set of conditionals in an information-theoretically optimal way. The problem of discovering conditionals from data can be seen as an instance of a general *inverse representation* problem. Within the CONDOR project (Conditionals - discovery and revison), methods and tools for discovery and revision of knowledge expressed by conditionals are developed. We will design, specify, and develop the ambitious CONDOR system using Abstract State Machines, based on previous experiences with the ASM approach and using tools provided by the ASM community.
(joint work with Gabriele Kern-Isberner)

Abstract State Machines Capture Parallel Algorithms

Andreas Blass
University of Michigan and (temporarily) Microsoft Research
March 4, 2002

This is a report on joint work with Yuri Gurevich. We give an axiomatic description of parallel, synchronous algorithms. Our main result is that every such algorithm can be simulated, step for step, by an abstract state machine with a background that provides for multisets.

Abstract State Processes

T. Bolognesi CNR, Istituto IEI, Pisa – bolognesi@iei.pi.cnr.it
E. Börger Dip. di Informatica, Univ. di Pisa – boerger@di.unipi.it
March 4, 2002

Process-algebraic languages and models offer a rich set of structuring techniques and concurrency patterns which allow one to decompose complex systems into concurrently interacting simpler component processes, abstracting however almost entirely from a notion of system state. Abstract State Machines (ASMs) offer powerful abstraction and refinement techniques for specifying system dynamics based upon a most general notion of structured state. The evolutions of the state are governed however by a fixed and typically unstructured program, called 'rule', which describes a set of abstract updates occurring simultaneously at each step (synchronous parallelism). We propose to incorporate the advantages offered by each structuring technique into one machine concept, and introduce to this purpose Abstract State Processes (ASPs), i.e. evolving processes (extended ASM programs which are structured and evolve like process-algebraic behaviour expressions) operating on evolving abstract states the way traditional ASM rules do. The ASP constructs are presented in Tables 1 and 2.

Table 1: Syntax of Abstract State Process

		Name	**Syntax of P**	**Alternative syntax and** *note*
Sequential ASPs	Nondeterministic and concurrent ASPs	Skip	skip	
		Assignment	$f(t_1, \ldots, t_n) := t_0$	
		Let	**let** $x = t$ **in** P	
		Conditional	**if** cond **then** P_1 [**else** P_2]	
		Sequentiality	$P_1 \gg P_2$	*Different from ASM construct* step *(or* seq*)*
		Process instantiation	$P_{id}(t)$	*where* $P_{id}(x) = P$ *is a process definition,* P_{id} *is a process name,* P *is a program*
		Choice	**choose**⟨Process set⟩	$\|$⟨Process set⟩
				choose x **with** cond(x) $B(x)$ *stands for* $\|\{B(x)\|\text{cond}(x)\}$
		Interleaving	**interleave**⟨Process set⟩	$\|\|\|$⟨Process set⟩
		Synchrony	**sync**⟨Process set⟩	$\|\|$⟨Process set⟩
				forall x **with** cond(x) $B(x)$ *stands for* $\|\|\{B(x)\|\text{cond}(x)\}$

Table 2: ASP semantics

Skip	$\langle \sigma,\ \text{skip} \rangle \Rightarrow (\varnothing,\ \text{nil})$
Assignment	$\langle \sigma,\ f(t_1, \ldots, t_n) := t \rangle \Rightarrow (\{(f\langle t_1[\sigma], \ldots, t_n[\sigma] \rangle,\ t[\sigma])\},\ \text{nil})$
Let	$\langle \sigma,\ \textbf{let}\ x = t\ \textbf{in}\ P \rangle \Rightarrow (\{(x', t[\sigma])\},\ P(x'))$, for fresh x'
Conditional	$\dfrac{t[\sigma] = \mathit{true} \wedge \langle \sigma,\ R \rangle \Rightarrow (u,\ R')}{\langle \sigma,\ \textbf{if}\ t\ \textbf{then}\ R\ \textbf{else}\ S \rangle \Rightarrow (u,\ R')}$ $\quad$ $\dfrac{t[\sigma] = \mathit{false} \wedge \langle \sigma,\ S \rangle \Rightarrow (u,\ S')}{\langle \sigma,\ \textbf{if}\ t\ \textbf{then}\ R\ \textbf{else}\ S \rangle \Rightarrow (u,\ S')}$ $\dfrac{t[\sigma] = \mathit{false}}{\langle \sigma,\ \textbf{if}\ t\ \textbf{then}\ R \rangle \Rightarrow (\varnothing,\ \textbf{if}\ t\ \textbf{then}\ R)}$
Sequential composition	$\dfrac{\langle \sigma,\ P_1 \rangle \Rightarrow (u,\ P_1'),\ P_1' \neq \text{nil}}{\langle \sigma,\ P_1 \gg P_2 \rangle \Rightarrow (u,\ P_1' \gg P_2)}$ $\quad$ $\dfrac{\langle \sigma,\ P_1 \rangle \Rightarrow (u,\ \text{nil})}{\langle \sigma,\ P_1 \gg P_2 \rangle \Rightarrow (u,\ P_2)}$
Process instantiation	$P_{id}(t)$ is equivalent to **let** $x = t$ **in** P, where '$P_{id}(x) = P$' is a process definition, P_{id} is a process name P is a program
Parameter computation (oper= *choose*, *synch*, *interleave*)	$\langle \sigma,\ \mathit{oper}\{P(x) \mid \text{Cond}(x)\} \rangle \Rightarrow (\{(x_1, v_1), \ldots, (x_n, v_n)\},\ \mathit{oper}\{P(x_1), \ldots, P(x_n)\})$ where $\{v_1, \ldots, v_n\} = \{v \mid \text{Cond is } \mathit{true} \text{ in } \sigma(x : v)\}$ and $x_1, \ldots, x_n$ are fresh and pairwise different.
Choice $\forall n \geq 2$, ASPs $P_1, \ldots, P_n$, $j \in \{1 \ldots n\}$	$\dfrac{\langle \sigma,\ P_j \rangle \Rightarrow (u_j,\ P_j')}{\langle \sigma,\ \textbf{choose}\{P_1, \ldots, P_n\} \rangle \Rightarrow (u_j,\ P_j')}$
Interleaving $\forall n \geq 2$, ASPs $P_1, \ldots, P_n$, $j \in \{1 \ldots n\}$	$\dfrac{\langle \sigma,\ P_j \rangle \Rightarrow (u_j,\ P_j')}{\langle \sigma,\ \textbf{interleave}\{P_1, \ldots, P_n\} \rangle \Rightarrow (u_j,\ \textbf{interleave}\{P_1', \ldots, P_n'\})}$ for each $i \in \{1, \ldots, n\} : P_j' = P_i$ **iff** $i \neq j$
Synchrony $\forall n \geq 2$, ASPs $P_1, \ldots, P_n$	$\dfrac{\langle \sigma,\ P_1 \rangle \Rightarrow (u_1,\ P_1'), \ldots, \langle \sigma,\ P_n \rangle \Rightarrow (u_n,\ P_n')}{\langle \sigma,\ \textbf{synch}\{P_1, \ldots, P_n\} \rangle \Rightarrow (\bigcup_{i=1,\ldots,n}\{u_i\},\ \textbf{synch}\{P_1', \ldots, P_n'\})}$

Definitional Suggestions for Computation Theory[1]

Egon Börger, Università di Pisa, Italy
boerger@di.unipi.it
March 6, 2002

For each of the principal current models of computation and of high-level system design, we present a uniform set of transparent easily understandable descriptions, which are faithful to the basic intuitions and concepts of the investigated systems. Our main goal is to provide a mathematical basis for technical comparison of established models of computation which can contribute to rationalize the scientific evaluation of different system specification approaches in the literature, clarifying in detail their advantages and disadvantages. As a side effect we obtain a powerful yet simple new approach for teaching the fundamentals of computation theory.

The systems we are going to study comprise the following:

- UML Diagrams for System Dynamics
- Classical Models of Computation
  - Automata: Moore-Mealy, Stream-Processing FSM, Co-Design FSM, Timed FSM, PushDown, Turing, Scott, Eilenberg, Minsky, Wegner
  - Substitution systems: Thue, Markov, Post, Conway
  - Structured programming
    - Programming constructs: seq, while, case, alternate, par
    - Gödel-Herbrand computable functions: Böhm-Jacopini Theorem
  - Tree computations: backtracking in logic and functional programming, context free grammars, attribute grammars, tree adjoining grammars
- Specification and Computation Models for System Design
  - Executable high-level design languages: UNITY, COLD
  - State-based specification languages
    - distributed (Petri Nets)
    - sequential: SCR (Parnas Tables), Z, B, VDM
  - Dedicated Virtual Machines: Active Database Machines, Data Flow (Neural) Machines
  - Stateless modeling systems
    - Logic based systems: axiomatic, denotational, algebraic
    - Process algebras (CSP, LOTOS, etc.)

[1] Draft of an Extended Abstract for an Invited Lecture at the FLoC'02 Workshop "Action Semantics and Related Semantic Frameworks", Copenhague, July 2002

Since we will use Abstract State Machines (ASMs) as modeling framework, a question to answer before proceeding is why we do not use (the proof for a version of) the ASM thesis which claims a form of computational universality for ASMs. The thesis as formulated in 1985 by Gurevich in a note to the American Mathematical Society [Gurevich85] reads as follows (where dynamic structures stand for what nowadays are called ASMs):

> Every computational device can be simulated by an appropriate dynamic structure—of appropriately the same size—in real time

For the synchronous parallel case of this thesis Blass and Gurevich [BlaGur01] (to appear in ToCL 2002) discovered postulates from which every synchronous parallel computational device could be proved by them to be simulatable in lock-step by an appropriate ASM. Why are we not satisfied for our purpose with the ASMs constructed by this proof?

The answer has to do with the fact that there is a price to be paid for *proving* computational universality from abstract postulates, covering a great variety of systems. On the one side, a feature the ASM method emphasizes is that of modeling algorithms and systems *closely and faithfully, at their level of abstraction*, laying down the essential computational ingredients completely and expressing them directly, without using any encoding which is foreign to the computational device under study. On the other side, if one looks for a mathematical argument proving from explicitly stated assumptions the computational universality of ASMs as claimed in the thesis, some generality in stating the postulates is unavoidable, to capture the huge class of data structures and of the many ways they can be used in a basic computation step, which for every proposed concrete system have to be derived (*decoded*) from the postulates.

The construction by Blass and Gurevich in op.cit., which in fact in a sense transforms an arbitrary synchronous parallel computational system into an ASM simulating the system step-by-step, depends on the way the abstract postulates capture the amount of computation (by every single agent) and of the communication (between the synchronized agents) allowed in a synchronous parallel computation step. The necessity to uniformly unfold arbitrary concrete basic parallel communication and computation steps from the postulates unavoidably yields some encoding overhead, to guarantee for *every* computational system which possibly could be proposed a representation by the abstract concepts of the postulates. As side effect of this— epistemologically significant—generality of the postulates, the application of the general transformation scheme to established models of computation may yield ASMs which are more involved than necessary and may blur features which really distinguish different concrete systems.

Our goal is that of *naturally* modeling systems of specification and computation, based upon an analysis of the characteristic conceptual features of each of them. We look for ASM descriptions for each established model of computation or of high-level system design, a propos including asynchronous distributed systems, which

- for every framework directly reflect the basic intuitions and concepts, by gently capturing the basic data structures and single computation steps which characterize the investigated system,
- are formulated in a way which is uniform enough to allow explicit comparisons between the considered classical system models.

By deliberately keeping the ASM model for each proposed system as close as possible to the original usual description of the system, so that it can be recognized straightforwardly to be simulated correctly and step by step by the ASM model, we provide for the full ASM thesis, i.e.including distributed systems, a strong argument which

- avoids a sophisticated existence proof for the ASM models from abstract postulates,
- avoids decoding of concrete concepts from abstract postulates,
- avoids a sophisticated proof to establish the correctness of the ASM models.

Since despite of listening to the specifics of each investigated framework and of tailoring the simulating ASM models accordingly we achieve a certain uniformity, this provides a mathematical basis for technical comparison of established system design approaches which we expect to

- contribute to rationalize the scientific evaluation of different specification approaches, clarifying their advantages and disadvantages,
- offer a powerful yet simple definitional framework for teaching computation theory.

For the modeling purpose, we generalize Finite State Machines (FSMs) to a class of Abstract State Machines (ASMs) which are tailored to UML diagram visualizable machines (introduced in [Boerger99] under the name of control state ASMs.)

References

[BlaGur01] A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. Technical Report MSR-TR-2001-117, Microsoft Research, November 2001.

[Boerger99] E. Börger. High level system design and analysis using abstract state machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.

[Gurevich85] Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, page 317, August 1985.

A precise semantics of UML State Machines: making semantic variation points and ambiguities explicit

Egon Börger, Università di Pisa - I
Alessandra Cavarra, Oxford University - UK
Elvinia Riccobene, Università di Catania - I
March 7, 2002

We present a rigorous and complete semantics for UML state machines. Our model *(a)* rigorously defines the UML event handling scheme making all its "semantic variation points" explicit, including the event deferring and the event completion mechanism; *(b)* encapsulates the run-to-completion step in two simple rules, where the peculiarities relative to entry/exit or transition actions and sequential, concurrent or history states are dealt with in a modular way; *(c)* integrates smoothly the state machine control structure with the data flow; *(d)* clarifies various difficulties concerning the scheduling scheme for internal ongoing (really concurrent) activities; *(e)* describes all the UML state machine features that break the thread-of-control; *(f)* provides a precise computational content to the UML terms of atomic and durative actions/activities, without losing the intended generality of these concepts, and allows one to clarify some dark but semantically relevant points in the UML documents on state machines.

Our rigorous semantics for state machines comes in the form of Abstract State Machines. We model the concurrent subactivities appearing in UML concurrent substates, and the internal activities of state machines, by multi-agent ASMs, given by a set of sequential agents, each executing its own rules.

Unless necessary to avoid inconsistencies, we do not take any position on which UML concepts or understandings of them are reasonable or desirable. Through our definitions we build a framework for rigorous description and analysis of logically consistent interpretations of the intuitions which underly UML concepts.

Concurrency and Refinement

Jim Davies
University of Oxford
March 7, 2002

The language of Communicating Sequential Processes (CSP) can be used to describe patterns of interaction: a process is a pattern of possible, communicating behaviour. This talk reviewed the essential aspects of CSP, and showed how the language could be used to specify a simple reactive system at five different levels of abstraction, each specification a refinement of the previous one. The meaning of process refinement was then discussed, and compared with the notion of sequential, data refinement.

Designing Software for Internet Telephony: Experiences in an Industrial Development Process

Giuseppe Del Castillo, Peter Päppinghaus
Siemens AG – ICM N PG U ID A 1
March 5, 2002

Experiences in the design and implementation of the CSCF, an entity within a mobile telephony core network, are reported. The CSCF (call state control function) works as a call controller within the IP Multimedia Subsystem (IMS) of the UMTS, a 3rd generation mobile communication system. The IMS architecture, unlike traditional telephony systems, is completely based on the Internet Protocol (IP). The IMS is being standardized by the 3rd Generation Partnership Project (3GPP). Within the IMS, the CSCF communicates with other entities by means of the Session Initiation Protocol (SIP), which is standardized by the IETF (Internet Engineering Task Force).

The design of the CSCF takes place within a strictly regulated industrial software development process. As a variation with respect to the standard development process, a high level executable model of the essential functionality of the CSCF was developed in the design phase. This model was successfully used to gain experience with the design in an early development stage and later taken as the basis for the C++ implementation of the CSCF product.

According to our experience, for the ASM method to become part of everyday industrial design practice, two requirements are seen as crucial: *(i)* seamless combination with object-oriented design methods like UML, and *(ii)* mature tool support.

Programming Languages from Evolutionary Point of View

Igor Durdanovic, NEC Research Institute
March 5, 2002

Inspired by natural evolution we in the machine learning community have tried to employ similar methods. In the last 6 years we have been working on evolving programs (and meta-programs) that solve "hard" combinatorial problems. We experimented with DNA-like assembler languages with limited pattern matching, typed Lisp-like symbolic expressions, Prolog-like pattern matching. Our experience shows that with limited computational resources that we have, low-level assembler like languages take to long to evolve and programs are hard to understand. Symbolic expressions and typing greatly reduce search space. However, the evolved programs are very brittle, i.e. small mutation can easily break them. We had best results with pattern matching languages, which makes a great case for Prolog as a programming language.

Our experience shows that having the "right" programming language is very important factor in software engineering. Programming languages that make use of pattern matching - functional languages, Prolog (unification) are very good for rapid prototyping and fare well in time-limited programming contests.

The advancement of computer power has enabled us to seek alternatives to human programming. Today we see many problems being solved by means of evolutionary algorithms instead of explicit programming. We expect to see an increase of use of evolutionary programming in industry as the computational power advances and field gets more mature.

Using ASMs for Controlling Technical Systems

Werner Gabrisch
University of Halle
March 4, 2002

Technical systems are safety critical systems. It's architecture is defined. There are a lot of methods used to describe the behavior of system parts. So it is difficult to define the behavior of a complete system.

All description methods can be defined by ASMs. We can handle systems if we are able to connect the descriptions of the parts. This can be done by using the protocol layers between the control layers, because all communication must base on it. They can be specified by ASMs, which are used by higher layers as submachins.

Description methods like statecharts can be structured in this way too. A statechart is an automata that controls actions (programs). The communication between automata and actions is specified by a set of given functions. These functions can used like a protocol layer. They define a basic signature for automata and actions.

As an experimental platform we are using a model Railway-system. In this system we will describe the behavior based on statecharts and its actions and generate a ASM of all. That must fit because the statecharts and the actions the same basic signature.

If we have defined a submachine, we will be able to check it. If we generate code based on this model. The specification of the model must be hold by the generated program. We can use the program as a function in higher abstraction levels.

Because of the generation is based on a hierarchical structure of signatures global states in different generation can't exists. These global states must be represented by dynamically functions.

We will also try to generalize this method to be able to describe large systems by a set of smaller models.

Resources, labels and Proofs

Didier Galmiche
LORIA - Universite Henri Poincare
Nancy, France
March 5, 2002

In this talk, we focus on resource logics for modelling systems and programs and on theorem proving in such logics. Resource is a basic notion in computer science and location, access to, consumption of resources are central concerns in the design of correct systems (like networks) and programs (acces to memory, data structures manipulation). We show how the use of algebra of labels solves problems like resource distribution in theorem proving in resource logics, the labels being the elements of resource models. We illustrate the main points of this approach within the logic of bunched implications BI and present labelled calculi that are complete w.r.t. resource semantics and provide proofs or countermodels of BI formulae. The relationships between resource semantics, labels and proof calculi are central in this presentation.

Using model checking to generate tests from ASM specifications

A.Gargantini†, E.Riccobene†, S.Rinzivillo‡
†DMI- University of Catania
‡DI- University of Pisa
March 7, 2002

In this presentation we tackle some aspects concerning the exploitation of Abstract State Machines (ASMs) for testing purposes. We present, for ASM specifications, a set of adequacy criteria measuring the coverage achieved by a test suite, and determining whether sufficient testing has been performed. We introduce a method to automatically generate from ASM specifications test sequences which accomplish a desired coverage. This method exploits the counter example generation of the model checker Spin. We use ASMs as test oracles to predict the expected outputs of units under test. We present some interesting results in evaluating the proposed method. We have measured, using the code coverage tool JCover, the coverage of code for SIS, Safety Injection System, provided by the test suite generated by our method. We found that statement and branch coverage were achieved. Moreover, our method has been compared with random generation of test cases. Finally, we present the generation of test cases for the OpenDoor example and we show how the non discovery problem for this example has been tackled and solved.

Engineering Concurrent Systems with Distributed Abstract State Machines

Uwe Glaesser
Technical University of BC Surrey
BC, Canada
March 5, 2002

Traditional engineering disciplines, like mechanical and electrical engineering, rely on well established instruments when it comes to specifying properties of technical systems with mathematical precision. In particular, the deliberate use of formalisms-e.g. such as the blueprints of mechanical engineering-is commonplace and well accepted. Software and systems engineering, however, deeply relies on informal requirement specifications even though establishing requirements is the first and most important step on the way from a fuzzy concept to a concrete implementation. Indeed there are good pragmatic reasons for deploying more appropriate instruments ([HM01], p. 171):

> ... software problems are the most difficult ones human beings have ever attempted to solve, and mathematics is the single most powerful intellectual discipline for problem solving. So it seems inevitable that the two will come together.

Furthermore, informal descriptions are not executable and as such provide only very limited support for experimental validation although this is ultimately needed for checking the accuracy of requirements against our intuitive understanding of the expected system behavior; indeed, it often is the only way to fully understand the implications of the specified requirements. Taking into account the practical needs of systems engineers and application domain experts, a good compromise for sharpening requirements into specifications means that *the "formal" sneaks in as we attempt to gain precision in natural language specifications*[HM01]. The talk exemplifies how the computation model of distributed Abstract State Machines and its underlying notion of partially ordered run, as defined in the Lipari Guide [Gur95], can be deployed for a gradual formalization of requirement specifications of complex distributed and embedded systems at a level of detail and precision as needed. Two recent industrial applications are presented, namely the Specification and Description Language - SDL standardized by the International Telecommunication Union - ITU, Geneva, and the Universal Plug and Play Device Architecture - UPnP [UPNP00] developed at Microsoft, Redmond. The first application is a comprehensive formalization of SDL based on a distributed real-time ASM [EGGLP01]. On behalf of the ITU and in close collaboration with ITU-T Study Group 10 (Languages for telecommunication applications), the dynamic properties of SDL have been formalized in terms of an SDL abstract machine in combination with an SDL-to-ASM compiler. In November 2000, this ASM model of SDL was approved as part of the current standard for SDL as defined by the ITU-T Recommendation Z.100

[ITU00]. The core of the SDL abstract machine is a distributed signal flow model defining the transportation of signals through an SDL system. The underlying model for delaying the transportation of signals also encompasses the behavior of SDL timers. Another application is a high-level executable specification of the UPnP protocol, the part of the UPnP architecture that ensures connectivity and interoperability among UPnP devices and control points, which also based on a distributed real-time ASM [GGV01],[GGV02],[GV02]. This modeling paradigm allows us to combine both *synchronous and asynchronous* execution models in one uniform model of computation. An executable version of the resulting behavior model is encoded in AsmL, the ASM Language, and comes together with a GUI allowing for the required control and visualization of simulation runs.

References

[EGGLP01] R. Eschbach, U. Glässer, R. Gotzhein, M. von Löwis and A. Prinz. Formal Definition of SDL-2000 - Compiling and Running SDL Specifications as ASM Models. *Journal of Universal Computer Science*, 7 (11): 1025-1050, Springer Pub. Co., 2001.

[GGV01] U. Glässer, Y. Gurevich and M. Veanes. Universal Plug and Play Machine Models. Foundations of Software Engineering, Microsoft Research, Redmond, Technical Report, MSR-TR-2001-59, June 15, 2001

[GGV02] U. Glässer, Y. Gurevich and M. Veanes. High-level Executable Specification of the Universal Plug and Play Architecture. In Proc. of *35th Hawaii International Conference on System Sciences* (HICSS-35), Software Technology Track, Hawaii, Jan. 2002.

[GV02] U. Glässer and M. Veanes. Universal Plug and Play Machine Models: Modeling with Distributed Abstract State Machines. To appear in Proc. of *IFIP World Computer Congress*, Stream 7 on Distributed and Parallel Embedded Systems (DIPES'02), Montreal, Aug. 2002.

[Gur95] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9-36, Oxford University Press, 1995.

[HM01] D. Hamlet and J. Maybee. The Engineering of Software: Technical Foundations for the Individual. Addison Wesley, 2001.

[ITU00] ITU-T Recommendation Z.100: Languages for Telecommunications Applications - Specification and Description Language (SDL), Annex F: SDL Formal Semantics Definition, International Telecommunication Union, Geneva, 2000.

[UPNP00] UPnP Device Architecture V1.0. *Microsoft Universal Plug and Play Summit, Seattle 2000*, Microsoft Corporation, Jan. 2000.

The SDL Virtual Machine

Reinhard Gotzhein, University of Kaiserslautern, Germany
March 4, 2002

The SDL Virtual Machine (SVM) is the core of the dynamic semantics of SDL, the Specification and Description Language. SDL is a design language for distributed systems, developed and applied in the telecommunications industry. The SDL Virtual Machine has been formalized based on Abstract State Machines by defining an SDL Abstract Machine (instruction set, architecture, resources), an SDL compiler, SVM agents, an SVM runtime library, and an SVM operating system. The SVM definition is part of the formal semantics of SDL, and has been approved as Annex F to the international SDL standard in November 2000.

Quantum Computing and Abstract State Machines

Erich Graedel (joint work with Antje Nowack)
Aachen University of Technology
March 8, 2002

We show that and how ASM can model quantum algorithms.

Abstract State Machine Language: Notes on Design and Implementation

Wolfgang Grieskamp, Microsoft Research, Redmond
March 5, 2002

The Abstract State Machine Language (AsmL) is a modeling language developed at Microsoft Research over the last few years. AsmL is build upon three foundations: the ASM paradigm for dealing with state, a rich set of mathematical data types like sets and maps with tailored notations, and the object and component model as found in the .NET framework. The language's notational style is oriented towards pseudo code. The language is a full consumer and provider of the .NET common language subset, faithfully reflecting concepts like classes, interfaces, delegates, events and so on. This allows AsmL on the hand to access all the framework functionality, on the other to be a modeling language *for* .NET. AsmL also introduces extensions compared to traditional ASMs. The concept of *partial updates* enables pointwise update on maps, sets and other data values. Partial update on maps replaces dynamic functions. The concept of *submachines* allows the description of one step of an ASM by means of several sub-steps. The language is implemented by compilation to C#. It has been bootstrapped, i.e. the compiler has been written in AsmL.

Explicating the intuition behind Abstract State Machines

Yuri Gurevich
Microsoft Research
March 4, 2002

The talk has two parts. In Part 1, we address the questions what is computation and what is algorithm. The analysis leads to our worldview: computation is state evolution, algoritms are abstractions of computers (where a computer is just the performer of the computation) that are state based. We discuss various challenges to this worldview. In Part 2, we consider distributed algorithms and explicate the notion of run of the Lipari guide.

UML/OCL and State Machines

Heinrich Hussmann
Dresden University of Technology, Department of Computer Science, Germany
March 7, 2002

This talk deals with the relationship between state machines and the Unified Modeling Language (UML) with its formal assertion language, the Object Constraint Language (OCL). Naturally, the talk concentrates on UML statecharts. In an introduction to UML statecharts, the wide range of possible semantics for UML statecharts is pointed out. In particular, the difference between statecharts as control machines and statecharts as protocol machines is explained. Control machines communicate with their environment in a usually asynchronous way by events and signals, whereas protocol machines describe the dynamic behaviour of a complex software component under usually synchronous operation calls. The need for a unifying semantics of the various smeantic variants is discussed. The OCL has close relationshops to UML statecharts, which are discussed in the second part of the talk. OCL expressions can be used for writing guards in UML statecharts. But moreover, the "oclInState" operation of OCL also makes it possible to express the semantics of a protocol machine in OCL, i.e. to translate a statechart into a set of OCL constraints. A planned extension of OCL for the version 2.0 of the UML standard, so-called OCL Message Expressions, are introduced, and it is shown how message expressions can be used to translate further aspects of statechart information into OCL. It is argued that in the long run it will be possible to define an adequate abstract semantics for UML statecharts indirectly through a formal semantics of OCL.

ASMs transformations and their correctness

Marcin Młotkowski
University of Wrocław
March 8, 2002

The Abstract State Machines can be used not only to study semantics of programming languages, but also to study some optimization techniques suitable to the given language. If a program P is represented by an ASM, then considered technique can be expressed by corresponding transformation of this ASM into another one. In order to prove its correctness equivalence between these two ASMs has to be proven. We present an algebraic approach to this problem. We define an equivalence relation between two algebraic structures, which represents "similarity" between these structures. Then we show that this equivalence is preserved during runs of both ASMs. Hence final algebras are equivalent, i.e. both ASMs give the same result.

A Logic for Abstract State Machines

Stanislas Nanchen and Robert Stärk
ETH Zurich
March 4, 2002

In this talk, we discuss a logic for Abstract State Machines. Unlike other approaches which are based on variants of Dynamic Logic, we propose new axioms and rules baeds on an update predicate on transition rules. In particular, we do not expect ASMs to be in normal form, but instead, we allow structural concepts for ASMs (sequential composition, recursive rule call).

The logic is an extension of FOL with a definedness predicate ($\mathrm{def}(R)$), an update predicate ($\mathrm{upd}(R, f, x, y)$) and a modal operator ($[R]\varphi$). With these new constructs, we are able to express the consistency of ASMs.

The logic is sound and, in the case of hierarchical ASMs (without recursive rule call), the logic is a definitional extension of FOL, and, by then, complete.

A consistent operation to compose Abstract State Machines and rigorous techniques to define, recognize and prove properties of distributed ASMs

Marianna Nicolosi Asmundo, Elvinia Riccobene
University of Catania
March 4, 2002

Modelling distributed systems in a coherent and rigorous way is a basic goal in Formal Methods. In this work we are addressing the problem for Abstract State Machines by giving an operation to integrate sequential ASMs together with some tests checking the consistency of simultaneous updates in the composed system and a method to define and prove safety and liveness properties of the system.

The composition operation takes as input a collection of sequential ASMs returning as output their composition (a multi-agents ASM). It is applied only after the execution of two tests checking the consistency of simultaneous updates. These preliminary tests are of syntactical nature. They simply do an inspection of the vocabulary and of the rules of the components. If one of them succeeds, the system is consistent with respect to simultaneous updates and the composition operation is successfully applied. If the preliminary tests fail, the composed system may still be consistent with respect to simultaneous updates but we cannot check it with an a priori test. Then we compose the system and check whether simultaneous updates of the same function are possible (test b.). If there is no run of the composed system allowing simultaneous updates of the same function then the system is consistent otherwise it is inconsistent.

Definition and proof of system properties is our concern as well. In this work we focus on the interfaces between the components and between the composed system and the environment describing their behavior, defining and proving their properties. In order to do that, we provide ASMs with the definitions of "interaction traces" and of "input/output traces" which describe the behavior of the ASMs on the interface hiding the internal behavior. System properties are defined as interfaces having their own vocabulary which is a collection of external and interaction functions and their set of traces or scenarios. This formulation allows us to define properties in a rigorous and precise way and to recognize safety and liveness properties by analyzing the structure of their traces. In particular it allows to apply classical and rigorous techniques to prove that a system satisfies a property like induction for safety properties and the construction of a progress function for liveness properties.

The Production Cell case study, already formalized in terms of multi-agents ASMs by Boerger and Mearelli, has been our first example of application and it gave good results. Now we are working on a consistent composition operation for distributed ASMs and investigating on a different kind of inconsistency regarding interactions between partially synchronized components which we called "synchronization inconsistency".

Deciding the Verification-Problem for ASMs via Model-Checking

Antje Nowack
LuFg Mathematische Grundlagen der Informatik
March 6, 2002

We focus on the decidability of the general verification problem:
"Given an ASM $\mathcal{A}$ and a temporal property φ. Does every computation of $\mathcal{A}$ satisfy φ?"

We identify a class of deterministic parallel ASMs and a fragment of temporal first-order logic such that the general verification problem is decidable for ASMs from this class and properties expressed in this fragment. The proof of the decidability is done via a reduction to the satisfiability problem of the clique-guarded monodic fragment of temporal first-order logic in $(\mathbb{N}, <)$ over finite, not expanding domains. This is known to be decidable.
Similarly (but partially easier), one can prove such a result for other classes of parallel ASMs and fragments of the monodic fragment of temporal first-order logic.

Time in ASMs - Some Problems and Solutions

Andreas Prinz
DResearch Digital Media Systems GmbH
March 6, 2002

Starting from the problems to cover time in the formal ASM semantics of SDL, various properties of time are analyzed. Following a proposal by Yuri Gurevich, a series of time axioms for SDL is provided and the corresponding semantics of ASMs is compared with this. Then some desirable extensions to the ASM time mechanism are given.

ASM in the classroom

Wolfgang Reisig
March 5, 2002

ASM can be motivated by its successful applications, as well as by Yuri's seminal theorem for (sequential) ASM. We followed the second way in a students course on Formal Methods. As an introductory example we started out with the well-known algorithm to construct a tangent at a circle. This algorithm has been chosen, as it has various , non-isomorphic realizations: a node is a pair (x_0, y_0) of real numbers or of integers $min \leq x_0$, $y_0 \leq max$, or a spot on a blackboard, or axiomatically given, or just a ground term, etc. Operations likewise vary over many realizations, adjusted to the representations of points, circles, lines, etc. Hence, this algorithm represents a typical, albeit very simple problem, that ASM is intended for. We emphasize that ASM is based on signatures (vocabularies): A signature Σ fixes the set of algorithms over Σ. To understand the borderline between algorithmic and non-algorithmic sets of behaviors, we consider a number of examples and counter-examples. For example, with a signature of one constant and one unary operation, the sequences (IN, i, suc) and (IN, $2i$, suc) are algorithmic, whereas (IN, 2^i, suc) is not.

Finally, we presented the observation that most non-trivial counterexamples follow the schema

$$\begin{array}{rccc} & S & \stackrel{\tau}{\longrightarrow} & S' & \\ h & \downarrow & & \downarrow & h \\ & Q & \stackrel{\tau}{\longrightarrow} & Q' & \\ = T & \downarrow & & \downarrow & t_{Q'} \neq t_{R'} \\ & R & \stackrel{\tau}{\longrightarrow} & R' & \end{array}$$

where h is a homomorphism and T a finite set of "witness", all interpreted equally on Q and R, but not on Q' and R'.

An Object Model Of Abstract Cryptography

Dean Rosenzweig, Neva Slani
University of Zagreb
March 7, 2002

The AsmL specification language suggests a model of objects and classes within ASMs, with a native notion of local state and accessibility. This allows a novel model of abstract cryptography, with the intended black-box properties of secrecy and indistinguishability, as postulated for usual term models, built in by construction. Due to ASMs, the model is rigorous, and due to AsmL it is executable as is. The Abadi-Rogaway mapping of abstract messages to probability ensembles relating abstract and computational indistinguishability carries over, and can be naturally extended from statics of messages and encryptions to dynamics of protocols. Dynamics of abstract and computational protocols can thus be directly related by correctness proofs in the same computational model.

Refinement of Abstract State Machines

Gerhard Schellhorn
Lehrstuhl fuer Softwaretechnik und Programmiersprachen, Institut fuer Informatik
Universitaet Augsburg
March 5, 2002

The talk describes a generic proof method for the correctness of refinements of Abstract State Machines based on commuting diagrams. The method generalizes forward simulations from the refinement of I/O automata by allowing arbitrary m:n diagrams, and by combining it with the refinement of data structures.

Modeling Mobility Management

Wolfgang Schönfeld
Fraunhofer IPSI
March 6, 2002

Mobility of a phone within a cellular network or of a computer within the Internet has to be managed by the respective network infrastructure: Routes to and from a mobile device have to be changed. In order to base mobility management of future cellular networks on IP, the Internet Protocol, various proposals like Mobile IP are currently being discussed. In this talk, we develop a simple model of mobility management, analyse handoff performance as well as its effect on payload. We also sketch the design of a network simulator for this kind of models.

ASM Concept in Refinement of Real-time Systems and in Verification

D.Beauquier, J.Cohen, A.Slissenko
University Paris 12, France
March 7, 2002

This talk unites two different subjects: compositionality of refinements of real-time systems and the verification of security of some cryptographic protocols. The particular feature of real-time refinements lies in real-time constraints. They impose some kind of "context-sensitiveness" of runs that drastically differs this case from classical refinements where runs are in some sense " context-free". As a consequence we loose many commutativity properties, in particular transitivity. Treating runs as evolving algebras helps to understand the difficulties related to compositionality and to develop notions that permit to give some conditions for compositionality. The second topic is the security of cryptographic protocols. We observe that the logical framework that we developed for the verification of real-time systems that uses ASM to embed programs into the logic works well also for the security of cryptographic protocols. Security of some (or many) session key distribution protocols (Needham-Schroeder, Otway-Rees etc), correct or bugged, is in a decidable class. For bugged protocols the decidability algorithm gives a description of possible attacks.

Continuous Software Engineering: Architectural Design of Evolutionary Software Systems

Asuman Sünbül
Kestrel Institute
March 5, 2002

The classical view of the software engineering discipline is revised under the circumstances, that today's large software systems often make it impossible to start the development of a system from scratch. In nearly all cases, already existing software components are re-used and assembled to form new software systems. The origin of software components can be manifold: they may be bought from outside vendors or taken from in-house software resources. Furthermore, software systems become obsolete during its operating time due to continuous changes concerning upgrading, local and global adjustments, improvement of software, or other kind of changes. These changes often lead to undesired side-effects. The original system structure cannot be kept, the system becomes obscure and untraceable. Thus, in order to avoid these unwanted effects, these changes must be part of a disciplined method capturing also the above mentioned long-term aspects of software development processes.

We suggest our Service Layer Model SLM approach which is focusing on component composition techniques. The semantics of the component model forming the technical basis for the approach is specified using the Abstract State Machines (ASM) method. Additionally, a component model input language (CMIL) is introduced as a user interface to the component model. In CMIL, system design processes are carried out using refinement techniques, especially by connector refinements. Therefore, the notion of abstract connectors is introduced as a mean to abstract from concrete connector specifications. The semantics of the refinement techniques is also fully formalized using ASMs. As a side effect, the formalizations contained in the work have directly been used to generate an executable version of the ASM rules using an ASM support environment. The generated code, the "CMIL-checker", has also been used to validate the examples and case studies presented in the work.

Information Services Semantics via ASM

Bernhard Thalheim, BTU Cottbus
March 4, 2002

Information Services are developed everywhere. Large sites have to be designed very carefully in order to be maintainable, be supportable by information systems and be extensible. Thus, a sound methodology is required. Beside methodologies or engineering approaches well-founded language must be developed. These languages should be able to cover various level of abstraction. The WebSiteLang developed at CottBus Tech aims in meeting these goals. Furthermore, the language can be entirely based on ASM. Thus, the operational semantics is provided by ASM's. The WebSiteLang has been challenged by a project aiming in development of TV-based set-top-box-based cable-channel-supported interactive information services. The great benefit of the semantic basis that has shown to be the most important advantage is the refinemendability.

Applications of ASMs in Microsoft: Modeling and Testing

Margus Veanes, Lev Nachmanson
Microsoft Research, Redmond
March 6, 2002

AsmL is an advanced ASM-based executable specification language that has been developed at Microsoft Research. It provides a modern specification environment that is object-oriented and component-based. AsmL is integrated into Microsoft Visual Studio, Word and COM.

AsmL can be used to model highly distributed architectures on any desired level of abstraction. Since AsmL models are executable they can be used for early design validation. Since they are easily readable they can be used for rigorous documentation. Since they have a precise semantics they can be used for static analysis and be used as test oracles. In this talk we discuss and demo our experience of using AsmL as a modeling tool of web service architectures. In particular we show parts of the Universal Plug and Play model of AsmL and we also demo the AsmL model of the business process description language Xlang.

We are working on an effort of making AsmL part of model-based testing in Microsoft. We discuss and demo how we plan to do this and what are the benefits. In particular, we are working on the integration of AsmL with an internal testing tool at Microsoft that is based on FSMs. The goal of this effort is to enrich the expressive modeling power of this tool and to avoid several shortcomings of FSMs that in general lead to state space explosion.

On ASM specifications for expression evaluation

Wolf Zimmermann
Universität Halle-Wittenberg
FB Mathematik und Informatik
March 7, 2002

We present a framework for formalizing the semantics of expression evaluation using Abstract State Machines. Many programming languages allow some nondeterminism for evaluating expressions. The semantics only have in common that arguments are evaluated before an operator is applied. However, programming languages usually restrict this most liberal evaluation order. For example, the expression evaluation may take into account short-circuit evaluation of boolean expressions which implies that right operands must not be evaluated before the complete left operand is evaluated. Our approach provides a generic expression evaluation semantics that only need to be instantiated adequatly. We demonstrate this approach by the example of Ada95, C, Java, and Fortran. (joint work with Axel Dold)