Abstract State Machines: An Overview of the Project

Yuri Gurevich
Microsoft Research, One Microsoft Way, Redmond, WA 98052

November 2003

Abstract

This is an extended abstract of an invited talk at the Third International Symposium on Foundations of Information and Knowledge Systems (to be) held in Vienna, Austria, in February 2004.
We quickly survey the ASM project, from its foundational roots to industrial applications.

0 Prelude

The ASM project started as a foundational investigation by this theorist in the mid 1980s at the University of Michigan. In the 1990s, a community of ASM-ers formed, see the ASM academic website [1], and several engines to write and execute abstract state machines were developed. Siemens was the first large company to use such engines. In 1998, Microsoft Research invited this theorist to build a group on Foundations of Software Engineering (FSE) and to apply the ASM theory. We sketch these developments stressing the foundational issues.

A draft of the talk, in the form of Power-Point slides, was written first; hence the choppy character of this hastily written extended abstract.

Many thanks to Andreas Blass and Jon Jacky for useful comments.

1 The original foundational problem

What is computer science about mathematically speaking? To elucidate the question, let's compare computer science to physics. The physicists use partial differential equations (PDEs) to model the physical world. What mathematics should play the role of PDEs in computer science?

The world of computer science is much different from that of physics. The precise state of a physical system is an abstraction, but the state of a computer (as a digital rather than physical system) is examinable. Much of physics is devoted to continuous processes where the very next state of the process does not exist. Computer scientists are interested primarily in discrete processes where the next state is well defined (unless the process stops). Here we concentrate on discrete processes exclusively.

Computer science isn't even a natural science: we study the artificial world of computers. Imagine intelligent visitors from a distant planet. Their mathematics and physics are probably similar to ours, but their computers and therefore their computer science may be vastly different. And yet there is something objective (and going beyond the classical analysis of which functions are computable) about computations that is worth exploring.

1.1 Dynamics

The classical mathematical approach to deal with dynamics is to reduce it to statics. Instead of analyzing motion directly, mathematicians analyze the history of motion. Your (for simplicity, ordinary) differential equations may speak about $dx/dt, dy/dt, \ldots$ where t is time, and so you seemingly study a process developing in time. But a simple coordinate transformation allows you to rewrite your equations in terms of $dt/dx, dy/dx, \ldots$, and now the process develops along a spatial line rather than the time line. This shows that time is just another dimension in the perfectly static history of the process. In that sense your (possibly most fruitful) analysis is sort of an autopsy.

This reduction of dynamics to statics does not come for free. We illustrate this on the case of a program with three integer variables x_1, x_2, x_3 that goes from the initial state X_0 to states $X_1, X_2, \ldots$. If we make the (logical) time explicit (so that the state at time t is X_t), then every x_i is a function $x_i(t)$ of t. For simplicity, we assume that every state X_t is uniquely defined by the values $x_1(t), x_2(t), x_3(t)$. Every $x_i(t+1)$ depends on X_t and therefore

on $x_1(t), x_2(t), x_3(t)$. Even if the original program was simple, the derived system of equations may be hard.

The ingenious mathematical analysis fakes a discretization of a continuous process and concentrates on the relation between the current state (at time t) and the "next state" (at time $t + dt$). We don't have to fake discretization because we deal with discrete processes to begin with.

Let's start our analysis of algorithms with algorithms that work in sequential time. The term algorithm is understood broadly; every computer system at any fixed abstraction level is an algorithm. A sequential time algorithm starts in some state X_0 and proceeds to states $X_1, X_2, \ldots$.

Postulate 1 (Sequential Time [10]) *The behavior of a sequential time algorithm is determined by the set of states, the subset of initial states, and the state transition function.*

Remark 1 Shouldn't it be the state transition relation rather than the state transition function? By default, an algorithm is deterministic. (One can argue that algorithms are intrinsically deterministic; see [10, Section 9] in this connection.) But nondeterministic algorithms (and nondeterministic abstract state machines) make appearance in the sequel.

A finite state machine is an example of sequential time algorithm. In general, a sequential time algorithm is a finite state machine or an infinite state machine. The computation theory offers us the universal Turing machine [13]. But it is clearly inadequate to describe arbitrary sequential time algorithms succinctly. Can one improve on Turing's machine?

1.2 Statics

It occurred to us early on that every static mathematic reality can be described as a structure in the sense of mathematical logic, that is a set with operations and relations. Next time you talk to a mathematician, ask him what he is working on. Whether he/she works with graphs or Banach spaces or whatever, it surely will be some kind of structures. This insight eventually gave rise to the Abstract State Postulate. We give here only an abbreviated version of the postulate.

Postulate 2 (Abstract State [10]) *The states of an algorithm are structures of a fixed vocabulary. ...*

The vocabulary is intrinsic to the algorithm itself and does not depend on the input or state. The current state contains all other information that the algorithm needs to continue the computation.

Remark 2 We use the notion of structure in a slightly unorthodox way. We presume that the base set of every structure contains the ideal elements `true` and `false` and that predicates are operations taking value in {`true`,`false`}. It follows that our states are algebras in the sense of the science of universal algebra.

2 Abstract state machines

What is the true state of a program in, say, the C programming language? Often, they tell you that the state is given by the values of its variables. This is not true. You need to know also the procedure stack and where the program counter is.

The Key Observation With fully transparent states (defined exclusively by the values of the variables), a simple programming language suffices to program transitions.

Note that the state of a C program can be made fully transparent by means of auxiliary variables [11]. The same applies to every other programming language.

The key observation led to the definition of abstract state machines, or ASMs.

Remark 3 We view a computation as an evolution of the state. According to the abstract state postulate and Remark 1, states are algebras. Hence the original name "evolving algebras" for abstract state machines.

We consider three categories of algorithms: sequential, (synchronous) parallel, and distributed. The definition of sequential ASMs was formulated in [8]. The definitions of parallel ASMs and distributed ASMs were formulated in [9]. Numerous examples of ASMs are found on the academic ASM site [1]. In the talk we illustrate ASM definitions by means of examples.

3 The foundational ambition of the ASM project

The ASM Thesis *Every algorithm is an ASM as far as the behavior is concerned. In particular the given algorithm can be step-for-step simulated by an appropriate ASM.*

This bold (impudent?) thesis was formulated in [9]. Recall that the notion of algorithm is understood broadly: every computer system at a fixed level of abstraction is an algorithm.

3.1 Theoretical confirmation of the ASM thesis

Intuitively, a sequential algorithm is a sequential time algorithm with steps of bounded complexity. In the presence of the sequential time postulate and the abstract state postulate, an additional *Bounded Exploration Postulate* expresses that the steps of any sequential algorithm have bounded complexity.

In [10], a sequential algorithm is defined as anything that satisfies these three postulates: the sequential time postulate, the abstract state postulate, and the bounded exploration postulate. Sequential ASMs are sequential algorithms of course. Two sequential algorithm are *behaviorally identical* if they have the same states, the same initial states and the same state transition function.

Theorem 1 (Sequential Characterization Theorem [10]) *For every sequential algorithm, there is a behaviorally identical sequential ASM.*

In [4], a parallel algorithm is defined as anything satisfying the sequential time postulate, the abstract state postulate, and several other postulates describing how the parallel subprocesses communicate with each other. The definition of parallel ASMs in [4] is a variant of that in [9]. In either version, parallel ASMs are parallel algorithms.

Theorem 2 (Parallel Characterization Theorem [4]) *For every parallel algorithm, there is a behaviorally identical parallel ASM.*

The problem of characterizing distributed algorithms by suitable postulates is open.

3.2 ASMs and hardware/software specifications

By the ASM thesis, ASMs are appropriate for modeling of arbitrary computer systems on given levels of abstraction. You can model existing or future systems; in other words, you can use ASMs to specify how hardware or software is supposed to function at a given level of abstraction. These specifications are executable. (Practical ASM languages typically use declarative means as well: preconditions, postconditions, invariants, and so on.) The executability of specification makes it much more useful. It allows you to address the following crucial questions.

1. Does the specification satisfy the requirements?
2. Does the implementation satisfy the specification?

3.3 Experimental confirmation of the ASM thesis

A substantial amount of experimental confirmation of the thesis is found at the academic ASM website [1]; see also books [5, 12]. In most cases people use ASMs not to check the thesis but to achieve their own goals; typically they use ASMs for modeling/specification purposes. But in the process they find out that ASMs suffice for their modeling purpose. In cases when Theorems 1 or 2 apply, a direct ASM simulation of a given piece of software or hardware may be more elegant than the generic simulation obtained from the proofs of Theorems 1 or 2.

One particularly impressive example of the ASM usage in academia is a large distributed ASM that gives the official dynamic semantics for SDL, the Specification and Description Language of the International Telecommunication Union [6]. Another impressive example is [12].

The use of ASMs at Microsoft is (very partially) reflected at [2].

4 AsmL, the ASM Language

ASMs are mathematical machines executable in principle. This is not good enough for applications. One needs practical engines to write down and execute ASMs. By the time I joined Microsoft, several ASM engines were in use. Siemens used ASM Workbench designed by Giuseppe Del Castillo at the University of Paderborn, Germany, as well as ASM Gopher designed by

Joachim Schmid and Wolfram Schulte at the University of Ulm, Germany. Matthias Anlauff designed XASM at the Technical University of Berlin, Germany; since then Matthias moved to Kestrel, Palo Alto, CA, and XASM became an open-source ASM tool. More information about these and other ASM tools is found at [1].

Currently, the most powerful ASM engines are those developed by the Foundation of Software Engineering group at Microsoft Research. One of them is called AsmL, an allusion to ASM Language. AsmL can be downloaded from the AsmL website [2] and used for academic purposes. The site contains various auxiliary materials.

4.1 Features of AsmL

AsmL has a strong mathematical component. In particular, sets, sequences, maps and tuples are available as well as set comprehension $\{e(x) \mid x \in r \mid \phi(x)\}$, sequence comprehension and map comprehension.

AsmL is fully object oriented.

The crucial features of AsmL, intrinsic to ASMs, are massive synchronous parallelism and finite choice. ASMs steps are transactions, and in that sense AsmL programming is transaction programming.

AsmL is fully integrated into the .NET framework which provides interoperability with great many languages and tools.

Literate programming via MS Word and automated programming via XML are provided. The demo, mentioned below, demonstrates literate programming among other things. The whole article [7] is in fact an AsmL document.

Here are some additional features of AsmL.

- Advanced type system: disjunctive types, semantic subtypes, generics,
- Pattern matching for structures and classes,
- Intra-step communication with outside world and among submachines,
- Reflection over execution,
- Data access, structural coverage,
- State as first class citizen,

- Processes (coming).

The AsmL compiler is written in AsmL.

4.2 Specifications vs. prototypes

It is often argued that specifications are mere prototypes. Of course, specifications are prototypes but good specifications are more than that. They present a consistent high-level description of the system abstracting away irrelevant details. They describe what might happen and what must not happen. And they are not quickly destroyed and thrown away; instead they continue to serve as important documentation.

Here is an example that makes this point; the example comes with the AsmL distribution. The task is to specify in-place sorting that proceeds one swap at a time and always advances. Here is an AsmL program that does that.

```
var A as Seq of Integer = [3,1,2]

Swap()
   choose i,j in Indices(A)
      where i<j and A(i)>A(j)
         A(i) := A(j)
         A(j) := A(i)

Sort()
   step until fixpoint
      Swap()
```

The program is self-explanatory with one exception: the last two lines of the `Swap` procedure are executed in parallel so that there is no need to save the value of `A(i)`. In AsmL, parallelism is a default; you pay a syntactic price for sequentiality.

The sorting algorithm of the program is not efficient but it is the most general algorithm for the purpose. Any other in-place sorting algorithm that proceeds one swap at a time and always advances is a specialization of our algorithm.

4.3 A demo

A demonstration of AsmL is planned for the talk.

5 Requirements, specifications, and implementations

Consider the development of a new piece of software (or maybe a new version of an old piece). A product idea gives rise to a (typically informal) description of the product formulating various requirements that the product is supposed to satisfy. This description is the starting point for writing a design specification. Eventually the specification is implemented.

5.1 Does the specification satisfy the requirements?

The question can be restated thus: how to debug the specification? Whether specification is declarative or executable, it is important that it is readable. But if the specification is executable, you can play out various scenarios. In the case of AsmL specification, given a few properties of the specification, the AsmL tool allows you to automatically derive a finite state machine that abstracts from other properties [7]. The finite state machine can be used to produce test suites and for model checking.

5.2 Does the implementation satisfy the specification?

The question really is how to enforce the specification? To make the problem a bit more concrete, imagine that our product is just an API, that is an application programming interface, that reacts to particular actions.

If the specification is deterministic, run a sequence of actions on the API specification and record the reactions; the result can be used as an oracle against which to test the implementation or implementations.

However, specifications tend to be highly non-deterministic. The sorting specification above is a good example. You cannot use it to produce an oracle for conformance testing. To deal with this more general situation, a different and much more subtle approach is being used by the group of Foundations of Software Engineering [3]. We plan to illustrate the approach in the talk.

6 Postlude

We hope that the story of the ASM project will support the maxim that there is nothing more practical than good theory.

References

[1] The ASM Michigan Webpage, `http://www.eecs.umich.edu/gasm/`, maintained by James K. Huggins.

[2] The AsmL webpage, `http://research.microsoft.com/foundations/AsmL/.`

[3] Mike Barnett and Wolfram Schulte, "Runtime Verification of .NET Contracts", *Elsevier Journal of Systems and Software* 65:3 (2003), 199–208.

[4] Andreas Blass and Yuri Gurevich, "Abstract state machines capture parallel algorithms," *ACM Transactions on Computational Logic* 4:4 (2003), 578–651.

[5] Egon Börger and Robert Stärk, *Abstract State Machines*, Springer, 2003.

[6] Uwe Glässer, Reinhard Gotzhein and Andreas Prinz, "Formal semantics of SDL-2000: Status and perspectives", *Computer Networks* 42:3 (2003), 343–358, Elsevier.

[7] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte and Margus Veanes, "Generating finite state machines from abstract state machines", ACM Software Engineering Notes 27:4 (2002), 112-122.

[8] Yuri Gurevich, "Evolving algebras: An attempt to discover semantics", in G. Rozenberg and A. Salomaa, Editors, *Current Trends in Theoretical Computer Science*, World Scientific, 1993, 266–292.

[9] Yuri Gurevich, "Evolving algebra 1993: Lipari guide", in E. Börger, Editor, *Specification and Validation Methods*, Oxford University Press, 1995, 9–36.

[10] Yuri Gurevich, "For every sequential algorithm there is an equivalent sequential abstract state machine", ACM Transactions on Computational Logic, vol. 1, no. 1 2000), 77–111.

[11] Yuri Gurevich and James K. Huggins, "The Semantics of the C programming language", Springer Lecture Notes in Computer Science 702 (1993), 274–308.

[12] Robert F. Stärk, Joachim Schmid and Egon Börger, *Java and the Java Virtual Machine: Definition, Verification, Validation*, Springer, 2001.

[13] Alan M. Turing, "On computable numbers, with an application to the Entscheidungsproblem", *Proceedings of London Mathematical Society*, series 2, vol. 42 (1936–1937), 230–265; correction, *ibidem*, vol. 43, 544–546. Available online at http://www.abelard.org/turpap2/tp2-ie.asp.