Logician in the land of OS: Abstract State Machines in Microsoft[*]

Yuri Gurevich
Microsoft Research
http://research.microsoft.com/∼gurevich

Abstract

Analysis of foundational problems like "What is computation?" leads to a sketch of the paradigm of abstract state machines (ASMs). This is followed by a brief discussion on ASMs applications. Then we present some theoretical problems that bridge between the traditional LICS themes and abstract state machines.

1 Introduction

This talk was prompted by Joe Halpern's invitation letter: "My hope this year is that the invited talks will showcase the relevance of logic to the rest of CS. It seems that some discussion of abstract state machines (and their potential impact on Microsoft) would be a great theme ..."

I always had a taste for foundational questions. That is why I went to logic (from algebra) in the first place. In 1982 Michigan hired me, a logician, on the promise to become a computer scientist. Contrary to mathematical logic where the foundational questions had been more or less settled, the foundational questions of computer science were wide open. What is it that we study in computer science? What is computation? What are the peculiar dynamic systems of computer science? Thinking about these questions, I arrived at the notion of abstract state machine (ASM) as a formalization of the notion of computer system at any given level of abstraction.

The operational approach of ASMs went against the pure declarative fashion of the formal methods of the time. Many formal-methods experts still think that any operational approach is necessarily low-level and that an executable specification is a contradiction in terms. But ASMs were successful in applications. The ASM community grew and with it grew the diversity of applications; see the ASM academic website [23] where you will find in particular a bibliography [13] and Egon Börger's surveys [11, 12]. While much of ASM activity takes place in academia, it is not confined to academia. Good ASM work has

[*]Sixteenth Annual IEEE Symposium on Logic in Computer Science, June 2001, Boston

been done in Siemens. There is an active ASM group in Microsoft. There are even two small ASM-based start-ups, http://www.modeled-computation.com and http://www.montages.com/.

The rest of this talk is organized as follows.

Section 2 A version of our original analysis of the fundamental questions mentioned above.

Section 3 A sketch of the ASM paradigm.

Section 4 A few words on what ASMs are good for.

Section 5 A few words on our Microsoft experience.

Section 6 Some theoretical problems related to ASMs.

Section 7 Postlude.

I showed a draft of this talk to my former student Quisani which resulted in some Q & A inserted in the text.

Acknowledgment

I am grateful to Andreas Blass, Mike Barnett, Uwe Glässer, Nikolai Tillmann and Margus Veanes for comments on this article which was written a little too quickly.

2 What is computation?

A computation can be defined as a run of a computer system. The notion of computer system should be general enough to account for future computer systems and for more abstract computations that you encounter, e.g., in the specification stage of software development. We proceed to make our notion of computer system a little more precise

2.1 Levels of abstraction

A computer system has a hierarchy of levels of abstraction. For example, you can view the execution of a C program on the level of the source program or on the level of the executable code. These are two different abstraction levels. Here we are interested in computations of a computer system with a fixed level of abstraction.

The need to fix a particular level of detail is well understood in software engineering. To this end, for example, APIs (application programming interfaces) enable the programmer to give precise syntactic information about a component — method names, typing information, etc. Typically the intended semantics is only hinted at. (And so you may want to use ASMs to fill in the gap.)

2.2 The program

A computer system is governed by a fixed program. Human society for example is not a computer system. The more focused theory of computer systems should be deeper than General System Theory.

A programmed system does not have to be closed. It can be highly interactive.

> **Q:** Is Internet a computer system in your sense?
>
> **A:** I guess this depends on the chosen level of abstraction. Even a complex system, like Internet, can be algorithmic on some levels of abstraction.
>
> **Q:** Shouldn't this apply to human society as well?
>
> **A:** You are right; it should.
>
> **Q:** Suppose that my program has loaded a bunch of classes from some library. Does this change the program of my computer system?
>
> **A:** Not necessarily. Again, this depends on the chosen level of abstraction. One possible view is this. Loading new classes changes only a part of your state; in particular the set of methods available to your program. The methods themselves can be seen as part of the active environment.
>
> **Q:** Maybe you should say "algorithmic system" rather than "computer system".
>
> **A:** Maybe. I used to say "algorithm" instead of "computer system" but there is a tendency to interpret the term "algorithm" too narrowly. Let's stick to the term "computer system" for the time being.
>
> **Q:** There are so-called non-von-Neumann systems which change their programs as they run.
>
> **A:** I saw some of them. Here is my understanding of how they work. There are fixed rules how to change the alleged program. Those rules constitute the real program. The alleged program is data.

2.3 The state

In general, a computer system is a dynamic system; it has a state that evolves in time.

> **Q:** Can a computer system be static? If yes, does it still have a state?
>
> **A:** Yes, and yes. Consider a sorting algorithm at the abstraction level where you abstract from everything except the input-output function that takes a given sequence to the sorted one. At that level of abstraction, no dynamics remains; the system still has a

state (including the sorting function) but the state does not evolve in time.

2.4 So what is computation?

Computation is evolution of the state.

Q: I guess you are talking about computations of a computer system at a fixed level of abstraction.

A: Yes, I am.

Q: This definition is not a mathematical definition.

A: Right. It is a philosophical speculation.

Q: I am skeptical about philosophical speculations. Give me one example of a philosophical speculation that proved to be useful.

A: Turing's speculative proof of his thesis [27].

3 The ASM paradigm

The notion of abstract state machine (ASM) formalizes our notion of computer system given at a fixed abstraction level.

The ASM Thesis Let A be any computer system at a fixed level of abstraction. There exists an abstract state machine B that simulates A step-for-step.

Q: How is this thesis different from Turing's thesis?

A: In many ways. In particular, a Turing machine would simulate A on the level of single bits while an ASM simulates A on the given abstraction level.

The "step-for-step" requirement is crucial. In distributed computing, typically only single steps are guaranteed not to be interrupted by other agents. If B simulates A step-for-step then it can substitute for A in distributed situations. Even if B makes only two steps to simulate one step of A, some other agent can intervene between the steps of B and mess up the simulation.

In [20], we proved the thesis for the case of sequential algorithms, more exactly for sequential-time algorithms with uniformly bounded parallelism.

Q: Is it a mathematical proof or another philosophical speculation?

A: It is a mathematical proof.

Q: How can you prove a thesis? The notion of sequential algorithms is informal.

A: We formalize the notion of sequential algorithms by means of three postulates: the Sequential-Time Postulate, the Abstract-State Postulate, and the Bounded-Exploration (that is stepwise uniformly bounded exploration) Postulate.

Work on more general versions of the thesis is in progress. Instead of defining ASMs here, we just sketch the ASM paradigm. The standard reference for the ASM syntax still is [19]; a new guide is in preparation.

Let A be a computer system at a fixed level of abstraction.

3.1 States as structures

States of A are first-order structures.

> **Q:** Why first-order? Why not second-order or higher-order?
>
> **A:** Second-order and higher-order and other kinds of logical structures can be viewed as special first-order structures. See for example article [10] where weak higher-order structures are treated as first-order structures.
>
> **Q:** Why should it be any kind of logic structure?
>
> **A:** The vast experience in applications of mathematical logic seems to confirm that any static mathematical reality can be adequately described as first-order structure.
>
> **Q:** It can be, I guess, adequately described in arithmetic.
>
> **A:** Arithmetization requires excessive encoding while structure representation is virtually free from encoding.

All states of A have the same vocabulary. The vocabulary reflects the invariant aspects of the algorithm. Further the base set of the state does not change during the evolution.

> **Q:** Many graph algorithms acquire new nodes as they run.
>
> **A:** But where do they take those new nodes from? We assume that the initial state has an infinite reserve of elements to be used as nodes or whatever. A special `import` (called also `create`) operator is used to fish out elements from the reserve and bring them to the foreground.

The set of states of A is closed under isomorphisms. Intuitively, isomorphic structures are representations of the same state. The details of representation should not matter.

> **Q:** If computation is state evolution and states are structures then computation is structure evolution.
>
> **A:** That is why abstract state machines used to be called evolving structures or evolving algebras.
>
> **Q:** Why algebras?
>
> **A:** An algebra is a structure whose vocabulary consists of function symbols. In logic, relations are different from functions because their

values live outside the structure. We tweaked the definition of first-order structures so that the Boolean values are always inside and thus our states are algebras.

3.2 State as a memory

In logic or algebra, structures are static. Our structures are dynamic. A state X is a memory (or store). If f is a function symbol of arity j in the vocabulary of X and if $\bar{a}$ is a j-tuple of elements of X then the pair $(f, \bar{a})$ is a *location* of X. The *content* of that location is the element $f(\bar{a})$.

3.3 Actions

An *atomic update* of a state X changes the content of one location of X. Since the vocabulary of A is fixed and the base set of the state does not change during the evolution, the set of locations does not change either. It follows that any transition from one state to another is characterized by an *update set*, a set of atomic updates.

The ASM syntax provides means to program atomic updates as well as various update sets. For example, if ϕ is a Boolean-valued term and R is an ASM rule generating an update set U at a state X then the rule `if` ϕ `then` R generates either U or $\emptyset$ over X depending on whether ϕ evaluates to `true` or to `false` over X.

Q: I guess state changes should respect isomorphisms of structures.

A: Of course. In [20], this is a part of the abstract-state postulate.

3.4 Runs

You have in general a number of computing agents executing their programs. It is convenient to think in terms of a global state. A move by an agent changes only a finite set of locations of the global state. Concurrent moves of different agents produce consistent changes. A run is a partial order of moves of various agents.

Q: Your global state is some kind of shared memory.

A: It is not a conventional shared memory.

Q: Consider a distributed system, say a network of computers. To make it more interesting, let us assume that different computers are located on different planets so that, by the relativity theory, the whole system does not have a global time. The computers exchange information via messages. Are there meaningful states of the system?

A: Yes, they are mathematical abstractions [19].

Further, agents themselves are represented in the state. The computation can destroy agents and create new ones. There could be various relations and functions involving agents [19].

3.5 ASMs and set theory

In a 1993 Dagstuhl conference, Andreas Blass said the following about formalizing algorithms as ASMs: “after a while it becomes clear that any ‘reasonable’ algorithm can be written as an ASM, just as any ‘reasonable’ proof can be formalized in ZFC.” This observation is analyzed and developed further in the chapter “ASMs and Set Theory” of his article “Abstract State Machines and Pure Mathematics” [4].

4 What are ASMs good for

The most obvious use of ASMs is to write executable specifications. Here is a sorting example.

You don’t need ASMs to specify that a sorting algorithm should sort. But suppose that, for some reason, e.g. security, you need that your sorting is in-place so that you only swap elements of the given array. Suppose further that you can do only one swap at a time. There are numerous ways to implement such sorting: quicksort, bubble sort, etc. Here is an ASM spec of in-place one-swap-a-time sorting. Suppose that a is an array with the set I of indices.

```
choose i,j in I with i<j and a[i]>a[j]
   do in-parallel
      a[i]:=a[j]
      a[j]:=a[i]
```

This rule is supposed to be executed over and over again until the computation halts (when the choice set becomes empty). This is the most general in-place one-swap-a-time sorting (such that every swap makes the array more sorted). You can employ various choice strategies and thus get more refined sorting algorithms; a refinement like quicksort is much more efficient than the spec. But the spec is executable as is, and appropriate ASM tools can execute it.

Q: Your notion of specification is very broad.

A: Yes. Whenever you have a pair of algorithms A and B so that B refines A, A is a spec for B. This includes the case when A is static and so the spec is declarative.

Q: Why is it important that specifications are executable?

A: Imagine that you have designed a cool product with many interesting features. Developers code it; this may take a while. Eventually testers may discover that the design was flawed and needs to be changed. You wish you could have played with your design before coding.

There are many more kinds of applications of ASMs; see [23] where you will find in particular a bibliography [13] and Egon Börger's surveys [11, 12].

5 ASMs in Microsoft

Jim Kajiya at Microsoft Research realized the potential of ASMs. In late summer of 1998, he invited me to start a new group, and I accepted. The ASM project had become more and more engineering, and I could use help. In addition, I was tired of analyzing old software and excited about the possibility to participate in the development of new software.

The new group was called Foundations of Software Engineering (FSE). By now we have a strong and busy ASM team that never seems to find time to dress up its outside window [14]. Our first priority is to develop a good tool to write and execute ASMs. A number of such tools have been developed in academia; see [23]. Two of these tools, ASM Workbench and ASM Gopher, have been successfully used at Siemens. However none of the tools was a good fit for the software development environment of Microsoft, and in particular for COM, Microsoft's Component Object Model [24]. We had to start from scratch.

Q: What is COM?

A: I quote from [2]: "Microsoft software is usually composed of COM components. These are really just static containers of methods. In your PC, you will find dynamic-link libraries (DLLs); a library contains one or more components (in compiled form). COM is a language-independent as well as machine-independent binary standard for component communication. An API for a COM component is composed of *interfaces*; an interface is an access point through which one accesses a set of methods. A client of a COM component never accesses directly the component's inner state, or even cares about its identity; it only makes use of the functionality provided by different methods behind the interface (or by requesting a different interface)."

The tool development in the group is headed by Wolfram Schulte, my first hire, who came to Microsoft in the summer of 1999 from the University of Ulm in Germany after completing his ASM-related habilitation there. Our main tool is called AsmL (ASM Language). It is an executable-specification language.

Q: What does it mean? Another high-level programming language?

A: It is a high-level programming language that implements the ASM paradigm. Accordingly it is highly parallel.

Q: What about that COM?

A: AsmL is COM compliant. You can specify a component, and the spec will have full COM connectivity. For example, a spec of a debugger may be much more concise and abstract than a real debugger, but it will be treated as a debugger by other COM components.

Q: Is AsmL optimized for efficiency or expressivity?

A: It is a pragmatic compromise but typically expressivity comes first.

Q: Are there product groups within Microsoft that use ASM technology?

A: Yes.

Q: Name one.

A: Universal Plug and Play.

This seems to be a wrong place to go into the details of our work. (A bunch of our papers should appear later this year in the Proceedings of ASM'2001 in Springer Lecture Notes in Computer Science. A few additional papers are headed elsewhere. We'll try to keep the website [14] current.) Instead let me share a few lessons that the group learned during its short existence.

- Verification isn't everything. Verification is great . . . when it is feasible. A spec is a basis not only for verification but also for testing, documentation, etc. Partial improvements can have a big impact
- Stay relevant. A spec must be testable and up-to-date.
- Integration is crucial. Without integration your tool may be useless. Integrate with the relevant developer environment (in our case, it is Microsoft Visual Studio). Integrate with the relevant run-time environments (in our case, they are COM, .NET and various libraries).

6 On ASM-related theoretical problems

I was asked more than once about ASM-related theoretical problems. Many appetizing foundational problems arise in applications. For example, what are objects and classes [21]? But let me keep closer to more traditional LICS themes (with hope to bridge between those themes and ASMs).

6.1 Fine complexity classes

The notion of polynomial time is very robust. The usual computation models including the Turing model give the same notion of polynomial time. In [22], we show that the usual computation models other than the Turing model give the same notion of nearly linear (that is linear times polylog) time. Linear time is much more sensitive to the choice of computation model, and there are numerous versions of linear time in use. One example is the linear time of computational geometry. The ASM model may have enough parameters to take care of all these versions of linear time — maybe. I did not investigate this.

In [6], we proved the linear-time hierarchy theorem for ASMs (that asserts that, as c varies, the classes of functions computable in time $c \cdot n$ form a proper hierarchy). As we wrote there, “One long-term goal of this line of research is to prove linear lower bounds for linear time problems”.

If you work with linear time and consider simulations, it is natural to require that simulation is lock-step, that is there exists a fixed k such that the simulator spends at most k steps to simulate one step of the simulatee. In [6], we used lock-step simulations with preprocessing to construct a diagonalizing machine and thereby proved the linear-time hierarchy theorem. Lock-step simulation deserves to be studied in its own right. To this end, Andreas Blass constructed a more involved diagonalizing machine that avoids preprocessing (unpublished).

It seems that the study of fine complexity classes was held back by the absence of an appropriate computation model. We hope that ASM can serve as such a model.

6.2 Computations with abstract structures

Contrary to conventional computation models, like Turing machines or random access machines, ASMs accept abstract structures as inputs. For example, an input could be a graph rather than a string (or adjacency matrix) representation of the graph.

> **Q:** Why is this important? Real computers do not accept abstract structures as inputs.
>
> **A:** You routinely abstract from representation details when you do specifications. But such abstraction is not confined to specifications. Suppose for example that I have a database in my computer and I ship it to you. You store it in your computer but the representation of the database in your computer will surely differ from that in mine. A query to database should not depend on the representation. To this end, popular query languages abstract from the representation, so that abstract databases are treated as inputs to queries. This is an important issue in database theory and practice [1].

In [18], I conjectured that there is no logic (or computation model) for polynomial-time computations with abstract structures; the conjecture implies P$\neq$NP and remains open.

> **Q:** I do not understand your conjecture. How can you quantify over logics?
>
> **A:** I assume that every logic satisfies some minimal requirements, in particular that the set of well-formed formulas is recursive.

In [10], ASMs were used as a computation model (and logic of a sort, called BGS) for a rich natural class of polynomial-time computations with abstract structures. Later Shelah proved the zero-one law for BGS [26, 5]. In particular, we show in [10] that counting is not available in BGS and that BGS cannot decide whether a bipartite graph admits a perfect matching. Later it was shown in [9] that if one adds counting to BGS then the perfect matching problem for bipartite graphs becomes expressible. It remains open whether the perfect matching problem for arbitrary graphs is expressible in BGS with counting. It is also open whether BGS with counting captures polynomial time. Other specific problems along these lines are discussed in [9]. In [16], ASMs were used to study logspace computations with abstract structures.

The complexity theory of computations with abstract structures deserves to be developed further.

6.3 Metafinite models

In [17], I preached finite model theory because many structures naturally arising in computer science are finite. In particular (the states of) relational databases are finite. But are they really finite in all cases? A database may use real numbers for example; where do those numbers "live"? Now consider the world of, say, the C programming language. It has arrays, records, arrays of records, records of arrays, and so on. It is convenient to model states of computer systems as infinite structures where only a finite part is active. To this end, we defined and studied metafinite structures in [15]. A metafinite structure has a finite primary part and possibly infinite secondary part.

In the case of a program state, the primary part reflects the active foreground and the infinite part reflects the passive background. One example is [10] where the background is the collection of hereditarily finite sets over the elements of the input structure. In general, background structures contain all the material (like maps of sets of sequences of maps) that the program may need. The notion of background was formalized in [7].

Metafinite structures are really ubiquitous and deserve more attention.

6.4 Interesting logics

What are logics appropriate to metafinite structures? That question has been addressed in [15]. Basically, you can quantify over the primary part only. The secondary part may have powerful operations. In the case of reals, for example, you may have multiset operations like sum, product, average, median. But you can't quantify over the secondary part.

The choice operator of ASMs (illustrated above, in Section 4) is typical for computer science. It is an independent-choice operator: different invocations of it produce independent choices. It differs from the epsilon operator of Hilbert, the classical choice operator of mathematical logic; different invocations of the epsilon operator over the same set produce the same result. In [8], we investigated the logic of the ASM choice operator. We found that this fascinating logic is much weaker than the logic of the epsilon operator.

There are other ASM-related logics waiting to be investigated. One example is first-order logic with `undef`, a special element that allows you to turn partial functions into total ones. This `undef` is different from `diverges` of recursive-function theory. An ASM program can refer to `undef` explicitly; in particular we allow tests like `x = undef`, and the equality `undef = undef` holds. This explicit use of `undef` makes the logic of `undef` more powerful than the other first-order logics of partial functions that I am aware of.

Until now we spoke about static logics. Once one introduces state transitions, new challenging issues appear.

What is the logic of the import operator mentioned above in Section 3? Unlike the choice operator, the import operator produces a different element every time it is invoked.

In the sequential-time case, an ASM program describes a single step (to be iterated). The state changes only at the end of the step, not at the middle. There are no side effects during the execution of one step. This feature of the ASM paradigm should allow one to develop clean logics to reason about at least one step of the program.

One may want also to use automated and partially automated systems, including model checking systems, to reason about the behavior of abstract state machines. The ASM community has some experience in this direction; see [25, 3] and the section on Mechanical Verification in [23]. We have a long way to go though.

7 Postlude

Logic that we use and apply in computer science is mathematical logic developed originally to build foundations of mathematics and to solve the problems in foundations of mathematics that arose in the beginning of twentieth century. Logicians distinguish clearly between syntax and semantics and strive to clarify both syntactical and semantical issues. Computer science applications of logic are much different from mathematical applications. Some of the strongest methods of mathematical logics, like the priority method and forcing, have not found direct applications in computer science. But the foundational tradition of logic is of great value to computer science at this stage of its development.

But computer science is not a purely mathematical discipline. It is an engineering discipline as well. In applications, it does not suffice to prove that the problem is decidable or even polynomial-time decidable. You may need a program that works reasonably fast on real computers. Some engineering compromises have to be made. It is not only syntax and semantics that we should worry about. It is also pragmatics. It may mess up your clean constructions, but it may also enhance them and make them work for the benefit of many.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A Case Study. In Y. Gurevich et al., editors,

Abstract State Machines: Theory and Applications (Proceedings of ASM'2000), volume 1912 of *LNCS*, pages 367–379. Springer-Verlag, 2000.

[3] D. Beauquier and A. Slissenko. A First Order Logic for Specification of Timed Algorithms: Basic Properties and a Decidable Class. *Annals of Pure and Applied Logic*. to appear.

[4] A. Blass. Abstract State Machines and Pure Mathematics. In Y. Gurevich et al., editors, *Abstract State Machines: Theory and Applications (Proceedings of ASM'2000)*, volume 1912 of *LNCS*, pages 9–21. Springer-Verlag, 2000.

[5] A. Blass and Y. Gurevich. Strong Extension Axioms and Shelah's Zero-One Law for Choiceless Polynomial Time. to appear.

[6] A. Blass and Y. Gurevich. The Linear Time Hierarchy Theorem for RAMs and Abstract State Machines. *Journal of Universal Computer Science*, 3(4):247–278, 1997.

[7] A. Blass and Y. Gurevich. Background, Reserve, and Gandy Machines. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL'2000*, volume 1862 of *LNCS*, pages 1–17. Springer-Verlag, 2000.

[8] A. Blass and Y. Gurevich. Logic of Choice. *Journal of Symbolic Logic*, 65(3):1264–1310, 2000.

[9] A. Blass, Y. Gurevich, and S. Shelah. On Polynomial Time Computation Over Unordered Structures. to appear.

[10] A. Blass, Y. Gurevich, and S. Shelah. Choiceless Polynomial Time. *Annals of Pure and Applied Logic*, 100(1–3):141–187, 1999.

[11] E. Börger. Why Use Evolving Algebras for Hardware and Software Engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *Lecture Notes in Computer Science*, pages 236–271. Springer, 1995.

[12] E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Proceedings of FM-Trends'98, Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer, 1999.

[13] E. Börger and J. K. Huggins. Abstract State Machines 1988–1998: Commented ASM Bibliography. *Bulletin of European Association for Theoretical Computer Science*, 1998. Number 64, February 1998, pp. 105–128.

[14] Foundations of Software Engineering Group at Microsoft Research. `http://research.microsoft.com/fse`.

[15] E. Grädel and Y. Gurevich. Metafinite Model Theory. *Information and Computation*, 140(1):26–81, 1998.

[16] E. Grädel and M. Spielmann. Logspace Reducibility via Abstract State Machines. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99*, volume 1709 of *LNCS*, pages 1738–1757. Springer-Verlag, 1999.

[17] Y. Gurevich. Toward logic tailored for computational complexity. In M. Richter et al, editors, *Computation and Proof Theory: Logic Colloquium 1983*, volume 1104 of *LNCS*, pages 175–216. Springer-Verlag, 1984.

[18] Y. Gurevich. Logic and the Challenge of Computer Science. In E. Börger, editor, *Current Trends in Theoretical Computer Science*, pages 1–57. Computer Science Press, 1988.

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

[20] Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[21] Y. Gurevich, W. Schulte, and M. Veanes. A Richer ASM Language (tentative title). In E. Börger and U. Glässer, editors, *Proceedings of ASM'2001*, LNCS. Springer-Verlag, 2001. to appear.

[22] Y. Gurevich and S. Shelah. Nearly linear time. In *Symposium on Logical Foundations of Computer Science*, volume 363 of *LNCS*, pages 108–118, Springer-Verlag, 1989.

[23] J. K. Huggins. Michigan Webpage on Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.

[24] D. Rogerson. *Inside COM.* Microsoft Press, 1997.

[25] G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM Case Study. *Journal of Universal Computer Science*, 3(4):377–413, 1997.

[26] S. Shelah. Choiceless Polynomial Time Logic: Inability to Express. In P. Clote and H. Schwichtenberg, editors, *Proceedings of CSL'2000*, volume 1862 of *LNCS*, pages 72–125. Springer-Verlag, 2000.

[27] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of London Mathematical Society (2)*, 42:230–236, 1936–37. Correction, *ibid.* 43, 544–546.