Abstract State Machines at the Cusp of the Millenium

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it (Visiting Microsoft Research, Redmond)

The ASM'2000 Workshop marks for the ASM method the transition from its adolescence to the maturation period. The goals which have been achieved open new frontiers and put us into the position to embark on new challenges.

1 The Start at the End of the Twentieth Century

We went a long way since the Spring of 1987 when Yuri Gurevich visited Pisa and, in a series of lectures on the fundamental problem of semantics of programming languages, presented the world première of the concept of ASMs (then called dynamic/evolving structures/algebras). He gave the main motivation: reconsider Turing's thesis in the light of the problem of semantics of programs. He illustrated his ideas with examples, in particular specifications of Turing machines, stack machines and some Pascal programs. He gave also proofs of simple properties of these programs. This material appeared a year later in [22]. It was preceded by the first appearance of the ASM Thesis, in embryo in a 1984 technical report [20], and fully spelled out in a notice presented on May 13 of 1985 to the American Mathematical Society [21]. It was accompanied by the first real-world application, namely the dynamic semantics of MODULA-2 [26], and shortly afterwards followed by the ASM treatment of concurrency used to define the semantics of OCCAM [27], which was presented by Gurevich in another series of lectures in Pisa in May 1990. Since then the concept of Abstract State Machines essentially remained stable [23, 24][1]and triggered hundreds of publications in various domains including finite model theory, complexity theory and numerous areas of applied computer science, in particular programming languages, database query languages, protocols, architectures and embedded control software [1].

The first attempts to put the bold ASM thesis to the test were focussed on the problem of the dynamics of programming languages known to us, and we came from a purely theoretical background and had no practical, let alone industrial, experience. What came out of that is a practical method which exploited ASMs for the development of a full-fledged refinement approach which enabled us to rigorously define and analyse the dynamic semantics of real-life programming languages and their implementation on virtual or real machines. By now, the

[1] The initially present construct to shrink domains, which was motivated by concerns about resource bounds, was abandoned because it belongs to garbage collection rather than to high-level specification. Some technical variation was later introduced concerning the treatment of non determinism and of inconsistent update sets.

covered programming paradigms include the paradigms of all the major modern programming languages. The method of constructing *ground models*, described in [5] where the name primary model (rather than ground model) was used, proved to be mature and was chosen for standardization purposes by the International Standards Organization, see [8, 9, 11, 12, 3], and by the International Telecommunication Union, as reported in [19] and in these Proceedings [18].

At the next step, the method was tried out for the specification and verification of machine architectures and protocols. Eventually this was followed by applications to software engineering. Here one starts by constructing a *ground model* for a proposed (or, in the case of reverse engineering, for an existing) software system. Through the entire design process, one refines and/or coarsens the models linking the high level models in a traceable and inspectable way to executable code; see the survey in [6].

The key for the surprisingly fast success of the method lies in (a) the two constituents of the notion of ASM, namely being *abstract* (Abstract State) and *operational* (Abstract Machine) (see the section *Abstract Machines + Abstract State = ASM* in [6] for the historical reconstruction of the confluence of these two concepts[2]) and (b) in the systematic way it offers for practical software development to *separate different concerns*.

The **abstract character** of ASMs allows one, on one side, to tailor the models to the needs or purposes of the design, and, on the other side, to make their rigorous analysis feasible. The latter is due to the freedom to use those proof methods which are appropriate for the present discourse. In other words, the abstraction mechanism, built into the notion of ASM, permits one to make *real* the old dream of well documented and controllable hierarchical system development. Use ASMs to do the following:

- Make the faithfulness of the models, with respect to the design intentions, checkable by direct inspection (falsifiable in the Popperian sense). This holds in particular in requirements engineering for the faithfulness of the ground model with respect to the informally given requirements. The faithfulness becomes checkable by the application domain expert once an ASM model is there (see [16]).
- Link, by hierarchies of stepwise refinements, the high-level definition in a transparent way to its implementation. Here each refinement step is supposed to reflect design decisions one wants to document for future use, e.g. for maintenance purposes or for changes by extensions and modifications.
- Make the (mathematical, possibly machine checked) justification of the correctness of a complex design feasible. ASMs offer the necessary rigorous framework for the analysis of run-time properties, at the appropriate level of abstraction, which allows one to prove that the implementation conforms to the high-level specification.

[2] Notice that the ASM abstract machine is different from the familiar abstract machines. It has built-in parallelism which allows one to abstract from irrelevant sequentialization.

The **operational character** of ASMs provides a sufficient basis for turning the definitions into executable models. These can be used for high-level validation of user scenarios prior to coding or of test beds by means of mental or machine driven experiments. This luckily breaks with the still widely held traditional view that specifications should be, or are by definition, non-executable.

The **separation of different concerns** is incorporated into the techniques provided by the ASM method for constructing different system views and linking them into hierarchies of system levels. Here are the major software development concerns we systematically separate and recombine, making *divide and conquer* the main methodical principle, and it led us very far.

- The separation of orthogonal design decisions is the most important one in practice. It is made possible by the most general abstraction and refinement capabilities of ASMs, and it is motivated by the necessity to keep the design space open as long as possible and to structure it, for "design for change" and for modular development.
- The separation of design from analysis corrects the long standing tradition of identifying "rigorous" with "formalized in logic". This tradition is one of the reasons why the so called formal methods have not really had a strong impact on practical software development.
- The separation, within the analysis, of experimental validation from mathematical verification is possible with ASMs because once the ASM models have been made executable, they can be used for simulation. The simulation of a higher-level model can be performed prior to writing the final code. It can also enhance testing the code, so that the correspondence of the implementation to the abstract specification can be checked.
- The separation of different degrees of detail within verification allows one to adapt the justification of a design to the current development stage. ASMs give the means to differentiate justifying a design to domain experts from justification in terms of mechanical reasoning systems. Further, one should distinguish between interactive logical systems and fully automated tools like model checkers or automatic theorem provers.
- The separation of mathematical concerns within verification. For example, one may want to split the proof of a property P for a complex system S into three steps, namely:
  - prove P for an abstract model M of S under an appropriate assumption A,
  - refine M to S so that S implements M correctly,
  - prove that S satisfies the assumption A.

  Experience shows that it is not only easier to prove properties for complex systems in this way, but this splitting of proof obligations often is the only known way to show that a run-time system works the way it is supposed to. A characteristic example are the ASM-based proofs of the correctness of compilation schemes [13, 10, 17] and the implementation of those schemes by provably correct real-life compilers, a problem which is addressed in these Proceedings. By the way, collecting such justificational evidence yields an interesting byproduct: a detailed analysis of the design itself.

Most of these lines of research and principles were present, in embryo, already at the first international ASM workshop held as early as 1994, as part of the IFIP World Computer Congress in Hamburg, Germany [30]. Although the applications which appeared there were limited by our academic experience and were largely motivated by an effort to critically test the ASM thesis, it is not incidental that right from the beginning, by making good use of the freedom of abstraction offered by the ASM concept, we were naturally led to "separate and combine" design levels, design and analysis, verification and validation, and degrees of detail in verification. Concerning validation, let us note that the first tools for making ASMs executable date back to 1990; see the Prolog-based compiler for ASMs Angelika Kappel developed in [29], to execute my ASMs for PROLOG [8], and the Michigan interpreter mentioned in [23][3].

Only 5 years later, at the ASM workshop which was held as part of the International Formal Methods Conference FM'99 in Toulouse, France[4], one can observe (see again the survey [6] for details) that

- the theory of ASMs is richly developed,
- the applications include industrially successful standardization and software engineering projects, some of which become publicly visible for the first time in these Proceedings [15, 4],
- there is a proliferation of different tools[5], most of them developed in academia, for both the experimental validation and the machine supported verification of ASMs, providing execution mechanisms for ASMs (via interpretation or compilation) and links to verification systems like PVS, KIV and model checkers.

All of these themes are reflected in the rich program of ASM'2000. These Proceedings, which constitute the first book entirely devoted to ASMs, document what has been achieved in the first decade after the formulation of the ASM concept. We see confirmation of our conviction expressed already at the

[3] Here is what Jim Huggins wrote to me on June 4, 2000, about the details of the history: "Yuri taught a course in programming language design at Michigan during the Fall of 1990. Of course, he introduced ASMs in the class. A very sharp young undergraduate named Ben Harrison wrote a bare-bones interpreter for ASMs in LISP in a weekend and distributed it to the class. Yuri was impressed enough with Harrison that he hired him to write a full-fledged ASM interpreter, this time in C. Ben built the core of the interpreter in May-June 1991. At that point the interpreter was handed over to me, and I worked on it for 3 years or so, finishing up the rest of the unfinished business at that time. In 1994 development was handed over to Raghu Mani, who worked on it for a couple of years...". Let me add that Raghu's task was to upgrade the interpreter for one-thread ASMs to multi-agent ASMs.

[4] In between, ASM workshops had been held in Paderborn (May 1996), Cannes (June 1997, June 1998) and Magdeburg (September 1998).

[5] When I was working on this introduction, a message from Prof. Igor Soloviev, of St.Petersburg State University, arrived: "One of my students, Andrew Usov, has written an ASM interpreter. It is implemented as an java-applet designed to run under an internet-browser (it has been tested under IE 5.0 and NS 4.7)."

first ASM workshop in 1994, namely that (paraphrased) "the extraordinary potential of the ASM method will change drastically the industrial future of formal specifications" [5, pg.393].

2 New Frontiers

The experience accumulated with the ASM concept, and with the method which has been developed for its use, did change the way we think about high-level software design and analysis. Now we have to actualize this vision to make it work for established software development disciplines, at a large scale. The achievements of the last decade open new frontiers and put us into the position to face the new challenges.

Through the extensive ASM modeling, validation and verification work of the past decade, the ASM thesis was experimentally confirmed. But this year brought us a theoretical explanation of the observed phenomenon, namely via a proof [25] that the sequential version of the thesis follows from three fundamental system theory axioms. Once established, the thesis allows one to draw conclusions of practical importance, as is illustrated by an example in these Proceedings [7]: the thesis guarantees that there is no loss of generality in substituting the fundamental but vague UML concepts of action and activity by the mathematically rigorous concepts of ASM step and ASM run. It seems that in UML [31, 2] the meanings of action/activity were intentionally left unspecified, namely to leave the space of possible implementations as open as possible. But this was achieved at the price of making it difficult to control the implications the concepts have in the context of the event-driven *run-to-completion* scheme, in particular concerning the possibly numerous and nested exit/entry actions, coming through interrupts, and concerning the launch and abortion of internal activities.

On the practical side we have to take advantage of the experience, acquired with building tools for executing ASMs, to develop an entire tool *environment* which is also industrially satisfactory. It has to support the different activities of defining, transforming (by refinements and by code generation) and analysing ASM models (by testing, via visualization supported simulation, and by verification). The tool environment has to enable us to capture the design knowledge in a rigorous, electronically available and reusable way, and to achieve this goal it must be integrated into established design flows and their tool environments. The integration potential of ASMs, as a universal model of computation which is well established by the accumulated experimental evidence and by the theoretical explanation we have by now for the ASM thesis, is helpful to capture the overall behavior of a complex system by combined use of whatever rigorous descriptions are appropriate and mandatory in established design approaches (static, dynamic, functional, state-based, object-oriented, etc.).

This is a difficult and probably long way to go, "a ridge walk between freedom and discipline, creativity and pattern oriented design, generality and specialization, expressability and limitations by tool support" [6]. But it is worth the effort.

Here are some among other challenging problems where I see a large potential for fruitful exploitation of the ASM method.

- If we succeed to construct paradigmatic and parameterized ASM components and to extract (de)composition techniques that can be made available in libraries, the "codeless" form of ASM programming will help porting application programs from one platform or language to another and can lead to fruitful applications for plug-and-play software technology.
- If we succeed to exploit ASMs for defining and implementing methods for generating test suites from high-level specifications, this will turn a dark and at present overwhelming part of software development into an intellectually challenging and methodologically well supported task of enormous practical value. Indeed using ASMs one can solve the crucial and essentially creative part of test case selection, given that this selection is driven typically by appplication domain expert knowledge and thus can be formulated using the ASM ground model. Similarly the ground model supports solving the oracle problem of testing: the expected output, which has to be compared with the execution output, can be defined using the ground model specification (which is independent of the programming language where the system will be encoded)[6].
- If we exploit ASMs to enhance current (mostly signature oriented) software architecture description techniques by adding to the structural definitions also relevant semantical content, we will solve a widely felt need for building reliably reconfigurable *conceptual* and *module interconnection architectures* [28].
- If we succeed to exploit the atomic transaction nature of the notion of ASM-step to model practically useful patterns for communication and synchronization of multi-agent ASMs, typically based on shared memory or on message passing, then we will contribute to solve a crucial problem of distributed computing.

On the theoretical side a model and a proof theory of ASMs are needed. We need definitions which capture and enhance the practical refinement schemes we have used with success for ASMs, together with useful proof principles which can be built into state-of-the-art mechanical verification systems (for some steps in this direction see the contributions to PVS and model checking in these Proceedings and [32, 14]). The proof theory we need should alleviate the verification effort encountered in practical applications, namely by offering structuring and layering of proof obligations which avoid the bottleneck of a priori fixed levels of overwhelming proof details. We need to find the right way to exploit the notion of monitored (real-valued) function for connecting the discrete ASM world to the continuous world of control theory. We need to help in building models for mobile computing. We badly need to extract the inherent object oriented features of ASMs, which are visible in the concept of ASM agents and of their

[6] A similar remark applies also to static testing (code inspection) where one has to formulate the properties to be checked.

state, to make them explicitly and syntactically available, adapted to established object-oriented programming techniques.

These Proceedings contain numerous contributions where the mentioned issues are raised and thus constitute a good point of departure to help solving the challenging problems which are waiting for us.

References

1. Abstract State Machines. `http://www.eecs.umich.edu/gasm/`.
2. *Rational Software Corporation, Unified Modeling Language UML, version 1.3*, 1999.
3. ISO/IEC 13211-1. Prolog-Part 1: General Core. In *Information Technology-Programming Languages*. International Standards Organization, 1995.
4. M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using ASMs at Microsoft: A Case Study. In *This volume*.
5. E. Börger. Logic Programming: The Evolving Algebra Approach. In B. Pehrson and I. Simon, editor, *IFIP 13th World Computer Congress*, number I (Technology/Foundations), pages 391–395. Elsevier, 1994.
6. E. Börger. High Level System Design and Analysis using Abstract State Machines. In D. Hutter and W. Stephan and P. Traverso and M. Ullmann, editor, *Current Trends in Applied Formal Methods (FM-Trends 98)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
7. E. Börger, A. Cavarra, and E. Riccobene. Modeling UML State Machines. In *This volume*.
8. E. Börger and K. Dässler. PROLOG. DIN Papers for Discussion. Report 58, ISO/IEC JTC1 SC22 WG17, April 1990.
9. E. Börger and B. Demoen. The view on database updates in Standard Prolog: a proposal and a rationale. Report 74, ISO/IEC JTC1 SC22 WG17 , February 1991.
10. E. Börger and I. Durdanovic. Correctness of Compiling Occam to Transputer Code. *Computer Journal*, (39(1)):52–92, 1996.
11. E. Börger and D. Rosenzweig. An Analysis of Prolog Database Views and Their Uniform Implementation. In K. Dässler and R. Scowen, editor, *Prolog. Paris Papers–2*, number 80, pages 87–130. National Physical Laboratory, Middlesex, July 1991.
12. E. Börger and D. Rosenzweig. The Mathematics of Set Predicates in Prolog. In K. Dässler and R. Scowen, editor, *Prolog. Copenhagen Papers–2*, number 105, pages 33–42. National Physical Laboratory, Middlesex, 1993.
13. E. Börger and D. Rosenzweig. The WAM–Definition and Compiler Correctness. In Ch. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, pages 20–90. Elsevier Science B.V./North–Holland, 1995.
14. E. Börger and J. Schmid. Composition and Submachine Concepts for Sequential ASMs. In P. Clote and H. Schwichtenberg, editor, *Gurevich Festschrift CSL 2000*, LNCS. Springer-Verlag, 2000. (In print).
15. E. Börger, J. Schmid, and P. Päppinghaus. Report on a Practical Application of ASMs in Software Design. In *This volume*.
16. E. Börger, J. Schmid, and E. Riccobene. Capturing Requirements by Abstract State Machines: The Light Control Case Study. *J. Universal Computer Science*, 2000. Special Requirement Engineering Issue, to appear.

17. E. Börger, J. Schmid, R. Stärk, and W. Schulte. *Java and the Java Virtual Machine*. Springer-Verlag, 2000. to appear.
18. R. Eschbach, U. Glässer, R. Gotzhein, and A. Prinz. On the Formal Semantics of SDL-2000: a Compilation Approach Based on an Abstract SDL Machine. In *This volume*.
19. U. Glässer, R. Gotzhein, and A. Prinz. Towards a New Formal SDL Semantics Based on Abstract State Machines. In R. Dssouli and G.v. Bochmann and Y.Lahav , editor, *SDL'99 - The Next Millenium (Proc. of the 9th SDL FORUM)*. Elsevier Science B.V., 1999.
20. Y. Gurevich. Reconsidering Turing's Thesis: Toward More Realistic Semantics of Programs. Technical Report CRL-TR-36-84, University of Michigan, Computing Research Lab, 1984.
21. Y. Gurevich. A New Thesis. *Notices of the American Mathematical Society*, page 317, 1985. abstract 85T-68-203, received May 13.
22. 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.
23. Y. Gurevich. Evolving Algebras: An Attempt to Discover Semantics. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 225–234. World Scientific, 1993. A reprint of the article in the Bulletin of the European Association for Theoretical Computer Science, Number 35 (1991), pp.71-82.
24. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
25. Y. Gurevich. Sequential Abstract State Machines Capture Sequential Algorithms. *ACM Transactions on Computational Logic*, 1, 2000.
26. Y. Gurevich and J. Morris. Algebraic operational semantics and modula-2. In Börger, E. et al., editor, *CSL'87, 1st Workshop on Computer Science Logic*, number 329 in LNCS, pages 81–101. Springer-Verlag, 1988.
27. Y. Gurevich and L. A. Moss. Algebraic operational semantics and occam. In Börger, E. et al., editor, *CSL'89, 3d Workshop on Computer Science Logic*, number 440 in LNCS, pages 176–192. Springer-Verlag, 1990.
28. C. Hofmeister, R.L. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.
29. A.M. Kappel. Implementation of Dynamic Algebras with an Application to Prolog. Master's thesis, CS Dept., University of Dortmund, Germany, November 1990. An extended abstract "Executable Specifications based on Dynamic Algebras" appeared in A. Voronkov (ed.): Logic Programming and Automated Reasoning, volume 698 of LNAI, Springer, 1993, pages 229-240.
30. B. Pehrson and I. Simon. *IFIP 13th World Computer Congress. Vol.I: Technology/Foundations*. Elsevier, 1994.
31. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.
32. G. Schellhorn. Verifikation abstrakter Zustandsmaschinen. Phd thesis, CS Dept., University of Ulm, Germany, 1999. For an English version consult www.informatik.uni-ulm.de/pm/kiv/papers/verif-asms-english.ps.gz.

To appear as introduction to the volume Y.Gurevich, M.Odersky, P.Kutter, L.Thiele (Eds): "International Workshop on Abstract State Machines ASM'2000", Springer LNCS, 2000