The ASM Method: a Cornerstone in Computer Science Education

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it
On sabbatical leave at SAP Research, Karlsruhe, Germany egon.boerger@sap.com

Abstract. The versatility and wide applicability of the Abstract State Machines Method for the design and the analysis of computational systems has not yet been fully exploited for teaching. We suggest to use it for introducing basic algorithmic concepts in a succinct and uniform way, which makes the definitions adoptable in traditionally unrelated courses, covering the full range of computing science curricula from computation theory to the engineering of software systems.

Through 15 years of successful applications of the Abstract State Machines method for high-level system design and analysis, ASMs became visible in practically every branch of the science of computing and found their way into industrial practice. The same cannot be said for the role ASMs play in the current practice of teaching, although they provide a simple foundation for the notions of pseudo-code and of virtual machines which pervade current curricula.

1 Reasons for the success of the ASM Method

To understand this gap let us step back and look for the reason of the success of the wide-spectrum ASM method, which has been developed in the 90'ies on the basis of the notion of ASMs through the work of numerous researchers who formed the still growing ASM community (see the ASM web site [43] and the annotated ASM bibliography in the AsmBook [20]).

The method starts with the mathematical definition of the **concept of ASMs** discovered by Gurevich in [39] and exploits its versatility in two directions, namely through the following two activities:

- *construction of faithful models* (so-called **ASM ground models**), formulated in terms of a given type of application or domain of discourse,
- *transforming models in a mathematically controllable way* by further detailing, leading through a sequence of so-called **ASM refinement** steps from ground models to machine code.

Construction of faithful models is supported by the possibility to fine-tune ASMs to express intuitive ideas about specific algorithmic processes or systems *directly.* 15 years of modeling experience with ASMs show that one can faithfully and rigorously capture one's algorithmic ideas by 'tailoring' their ASM descriptions in terms of the investigated application domain (universe of discourse), 'at the given level of abstraction', avoiding any deviation from the problem analysis which could result from an encoding into *a priori* determined structures (of data and operations). This versatility of ASMs constitutes the content of Gurevich's ASM Thesis—formulated for the first time in [38], ten years before the definition of ASMs in [39]—and of the convincing epistemological analysis of this thesis provided 15 years later for the sequential case [40].[1] The practical exploitation of this adaptability of the language of ASMs— to describe the mathematical algorithmic content underlying real-life phenomena by a rigorous dynamic model, which provides the basis for a rational justification of its 'correctness' with respect to the original intentions—led me at the beginning of the 90'ies to the formulation of the *ground model* concept [9,11]. ASM

[1] I do not include here the analysis of the concept of bounded synchronous parallelism in [8] because of some reservation about the 'naturalness' of the concepts brought forward through that investigation; see the discussion in [14, Sect.1].

ground models constitute a mathematical form of accurate high-level system blueprints, which play a central role in requirements engineering, software contract formulations, standardization efforts and the correctness analysis for complex algorithms, protocols and processes in various domains: technical (device or plant) control, administrative, economical (business processes or web services [57,31,32,33,29]), medical, biological, etc. The wide range of processes one can capture with ASM ground models is well illustrated also in these Proceedings where the use of ASMs is reported to model such different things as urban crime patterns [23], fault and event handlers in web services [30,7], data warehouses [53], an interpreter for the extension C#2.0 of C# [44], the runtime environment CLR for executing .NET applications [37], security protocols [41], GUIs [48] and object-oriented programming design patterns [15].

Mathematically controllable model transformations leading from ground models to machine code are supported by the comprehensive mathematical *ASM refinement* notion, to whose definition I was led again by practical concerns and experience [9,12], generalizing the great variety of refinement definitions in the literature. The definition of ASM refinement puts the abstraction potential of ASMs to use for a combination of static data refinement and dynamic action refinement in a coherent framework, which supports the piecemeal introduction and documentation of relevant design decisions in such a way that even for complex real-life systems, their mathematical verification and experimental validation can be achieved. The arguably most general ASM refinement definition supports as special case also the *incremental program development and verification* endeavor,[2] which has been advocated in the structural programming program [58,28,25] and has been studied intensively in numerous algebraic, set-theoretic and logical specification and verification approaches [59,6,26,34,27]. The ASM refinement notion also generalizes the standard paradigm of implementing abstract methods (interfaces) by (sub)classes, as used in particular in object-oriented programming (see the discussion of some typical object-oriented design patterns in [15]).

The success of the ASM method is due to the possibility of systematically exploiting ASMs in those two directions. In system development applications it allows one to link the typically not-computer-centric application-domain-view of a system, determined by objects and concepts related to the real world surrounding the to-be-constructed or to-be-analyzed system, in a coherent and mathematically checkable way to the implementation view of the finished computer-based system. In design and analysis of algorithms it allows one to split the verification of properties into two usually simpler tasks: a) the proof of the properties from some axioms for an abstract model, b) a proof that the refinement of the abstract model satisfies the axioms.

2 Transfer of the ASM method to teaching

The double use of ASMs analyzed above, as ground models and as refinements of abstract models, is what can be exploited with advantage also for teaching computing science subjects.

Lecturers have to teach in one way or the other both of those two activities involved in algorithm and system design and analysis. This holds even in the traditional view, shared by many practitioners, where besides the requirements there is nothing else than their implementations; in fact a) incomplete or not sufficiently precise requirements documents have to be turned anyway into some rigorous form (read: a ground model), whether right from the beginning or piecemeal during the coding work, and b) every implementation is a special case of refinement, namely of the requirements into code (read: a target programmming language model). One can exploit ASMs and their refinements as a once-explained, simple and uniform concept of a computational system and of its piecemeal definition by successive detailing (in a full system development cycle this also covers later extensions), in courses whose themes range from purely theoretical issues—e.g. classical computation and complexity theory—over algorithms, protocols, specification and programming concepts to requirements capture and high-level hw/sw system design and analysis.

First of all this helps to not repeat again and again, in varying notations, the same fundamental computational concepts. For example, standard textbooks on the theory of computation define the

[2] See Schellhorn's detailed technical analysis in [50,51,52].

notion of computation and of what is computed or generated for finite state machines and various extensions (PDAs, Turing machines, tree automata, timed automata, alternating TMs, interactive TMs, to mention only a few of those commonly taught). Then later these concepts are repeated for data flow machines, B machines, Petri nets, in applied courses for all sorts of virtual machines, in software engineering courses for UML activity diagrams, Message Sequence Charts, UML state machines, etc. In reality all these classical concepts are instances of a few easily definable ASM patterns (for the details see [14]). But more importantly the ASM method helps to extract the underlying computational concepts in an explicit uniform manner, so that where needed to capture standard variations they can easily be refined (in fact typically just instantiated by further detailing of some relevant parameters and constraints).

In [13] we have outlined an ASM-method-based approach to teaching standard courses in computer science curricula. It is characterized by using throughout ASMs as basic language and the definition of ground models and their refinements as specification and implementation frame. The simplification one obtains in treating standard matters in this uniform but mathematically precise way is noteworthy. It is a delight for the lecturer to experience how quickly one can proceed in class from simple basic theoretical concepts to their application in real-life case studies students willingly accept as relevant. This builds upon the impressive variety of ASM applications, which have been elaborated in practically every corner of computing science (for a survey see the historical Chapter 9 of the AsmBook [20]) and are mostly available at the ASM web site [43]. The frugal character of ASMs as sets of conditional commands, rule-based systems with rules (pseudo-code) of form

if *this-is-the-case* **then do** *that*

or UML activity diagram rules or event-B transitions [1] of form

if *event* **then** *action*

helps to keep the interest of students by going after some short preparations (to define the notions of ASM and ASM refinement) into a great variety of challenging material. We avoid to teach theory for its own sake, avoid over-formalizations and reserve the use of mathematical (or logical) techniques to those moments where they are indispensable for a system design and analysis discipline to be called trustworthy.

Introducing ASMs from Scratch. Here is an example. In [16][3] we start right away from the notion of Finite State Machine and extend it to mathematically capture the nowadays omnipresent concept of Virtual Machines [47]—by simply replacing ‘Finite State’ by ‘Abstract State’, adopting for this purpose the *elementary diagram* view of Tarski structures. This view is well known from logic (more precisely from model theory) and easily explained to computer science students in terms of an abstract (platform and language independent) memory concept, thus circumventing the need for preliminaries from logic: consider states as defined by the values residing in abstract updatable memory units, otherwise stated by the content of a *set of abstract updatable locations*. In this view, locations are containers holding values of whatever kind, whether atomic or structured: objects, sets, multisets, functions, tables (as used successfully for example in the SAP programming language ABAB [46]), networks, processes, whatever one wants the system to deal with. State changes are simply updates of the value content of (some of the) locations via assignments. It now suffices to

- group value sets into tables [49,10,42],
- guard the updates by conditions (as in Dijkstra’s guarded commands many students know)

to obtain the UML activity diagram transitions (more precisely their synchronous version, which coincides with basic ASMs [17] and with event-B transitions [16]), interpretable as general form of virtual machine programs.

One can then explain the well-known *classification of ASM locations* (and functions or tables) as a means to identify modular state and machine components, which can be dealt with independently,

[3] See slides at `http://se.inf.ethz.ch/teaching/ws2004/0239/slides/AsmMethZh04.PDF`

whether in an operational or in a declarative manner. It can also serve to motivate the special **choose** notation for selection (read: abstract scheduling) functions, as an attempt to make some inherent non-determinism explicit. That leads naturally to also introduce its pendant, the **forall** construct—at this point the students have learnt the full ASM concept. Asynchronous ASMs to capture distributed computations are a matter of appropriately extending the notion of sequential run to distributed runs; there are various ways to do this, with or without specific scheduling concepts as for example interleaving (e.g. along the lines followed for Event-B models [1,4]), but in all cases without changing the concept of ASMs.

Training construction, refinement and analysis of models constitutes the rest of the course, going through appropriate case studies many of which are available in the ASM literature—they should be chosen on the basis of the students' concrete knowledge of specific systems or of their concrete interest to build a particular system. The achievements one can obtain by teaching these three activities are enormous.

- Teaching *how to construct ground models*. If you choose interesting or practically relevant systems or algorithms for this modeling activity, an immediate yield is that your students study and learn those systems or algorithms in detail, respectively that they learn to accurately analyze and document the system or algorithm to be realized. A side effect of this is that students learn to critically read system descriptions and to detect inaccuracies in documents provided to them. A longer lasting positive effect is that through this formalization activity students learn an ability that is essential for good requirements capture and system design, namely to *formulate exactly* what one wants to express and to lay the text for a rational, objectively measurable evaluation by inspection methods. 'Exactly' here means the following three things.
  - Precision: whatever is stated has to be stated accurately, just as much as needed to avoid undesired ambiguities in the resulting model. If something is intendedly left open, this fact and the degree of freedom one wants to leave (for successive refinements leading to an implementation) has to be circumscribed in a clear manner. We exploit here extensively the freedom ASMs offer to fine-tune the definition of auxiliary functions and predicates, combining more or less loose 'declarative' (logico-algebraic, axiomatic, equational etc.) definitions with constructive ('operational') ones (using modules and submachines), supported by the well-known classification of ASM locations and functions (into static/dynamic and monitored, controlled, shared, output).
  - Completeness: everything needed for the desired system has to be stated, to avoid that system features deemed as essential are not visible right from the beginning and thereby not controllable during the development process (traceable to code). System blueprints commit the involved parties of the software contract (read: the domain expert and the system designer) and thereby have to be complete to prevent later arguing about the meaning of the contract.
  - Minimality: nothing not-needed should be stated, to keep the resulting specification as simple as possible (this property often decides upon the understandability and thereby the usefulness of the ground model) and not to unnecessarily restrict the implementation space by too early design decisions.
- Teaching *how to successively add details*, starting from ground models, to reach compilable code through chains of refined models, each reflecting some relevant orthogonal design decision. In this way students learn to build models and code 'for change'; the abstractions provide interfaces where one can hang in different refinements.
- Teaching *how to analyze (simulate and verify) models and their refinements* rigorously. The mathematical character of ASM models makes both their execution (by Gedankenexperiment or by machines) and the mathematical verification of their run-time properties possible. This is the place where students can learn that contrary to a widely held view, the application of mathematical techniques can be made possible for complex systems, if one chooses the right level of model detailing (abstraction) where such methods can be used effectively. The ASM method provides the possibility to turn a system into a mathematical object one can investigate

with mathematical methods. The importance of and the price for verification besides the validation efforts can be experienced by letting the students detect through reasoning methods (using their own heads or with support from theorem proving tools) some gaps which went uncovered through testing. Very interesting and didactically carefully thought through material for tool supported system verification can be found in Abrial's work on Event-B models, e.g. [5,4,3,2], which can be adapted to the ASM framework using the precise relation between Event-B models and a rather large class of ASMs defined in [16]. It is a superset of Gurevich's sequential ASMs [40], due to the underlying interleaving concept for modeling distributed computations; due to the interleaving mechanism it is a subset of asynchronous ASM.

In doing this one can methodologically cover the full range of software engineering activities, leading from capturing verbal requirements to coding and verifying or validating (simulating) models at any relevant level of abstraction.

Material for Lecturing. For introductory courses the AsmBook [20] provides the needed definitions and illustrates them by small examples taken from the ASM literature (slides for lecturers are available on the AsmBook web site `http://www.di.unipi.it/AsmBook/`). Advanced courses on principles of programming languages—dealing with implementation and verification issues, covering semantics, compilation, bytecode verification and concurrency—have been successfully based upon the Java/JVM case study, which exhibits the full range from ground model to executable code with verification and validation of relevant system properties. It is publicly available in [56], slides for lecturers are available on the web site `http://www.di.unipi.it/~boerger`. The method explained in that book has been successfully reused for C# and experienced already repeatedly in class. The detailed documentation is contained in the papers [18,55,21,54,36,35,37,44,45], slides for lecturers are available on Robert Stärk's web site `http://www.inf.ethz.ch/~staerk/mcomp04/`.

3 Some Challenges

In addition to the growing number of interesting ASM case studies one finds today in the literature and on the ASM website [43], we need more such full-fledged books and dedicated course material, written on different subject areas and putting ASMs to use in those areas: complexity theory, algorithms and data structures, compiling methods, embedded (real-time) systems, requirements capture, performance analysis, business process management, testing methods, design and analysis of circuits, hardware, protocols, web services, architectures, etc. Similarly to the decisive role extensive applications played for the development of the ASM method—namely for the definitive formulation of the three concepts of ASM [39], ASM ground model [11] and ASM refinement [12]—also teaching the ASM method for high-level system design and analysis is best performed by fully carrying out in class challenging and representative case studies, leading from the IP level—dealing with the intellectual conception of a system—to the level of its realization and including (as much as reasonable) the documentation, the verification and validation of every relevant design and implementation step. This is a challenging task the ASM community should address.

This is a good place to refer to another challenging idea I have mentioned in various lectures since 1999, when it has been suggested to me by my son working in biology [22], namely to use ASM models in computational biology. The following suggestion is quoted from an exchange of messages with Clote [24]:

> Increasingly, especially for statistical classification problems involving amino acid or nucleotide sequence information (e.g. prediction whether a protein is a G-coupled protein receptor, whether a disulfide bond exists between 2 cysteines when given their neighborhood residues, etc.), one finds applications of
>
> - neural nets,
> - hidden Markov models,
> - support vector machines, etc.
>
> It might be possible to envision a class of bioinformatics problems, possibly involving optimization, for which the Abstract State Machine paradigm is useful. In particular,

what strikes me as most promising is to model signalling networks or to model metabolic pathways in a time-dependent manner.
There have been applications of stochastic Petri nets for the previous two problems, so presumably some form of stochastic Abstract State Machine might be worth looking at.

Except for the modeling of neural nets by ASMs in [19], it seems that up to now Clote's idea has not been investigated. We hope someboy will take it up in the near future.

References

1. J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *Proc. 1st Conf. on the B Method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.
2. J.-R. Abrial. Event based sequential program development: application to constructing a pointer program. In *Proceedings FME 2003*, pages 51–74. Springer, 2003.
3. J.-R. Abrial. Discrete system models. Version 2, September 2004.
4. J.-R. Abrial. Event driven distributed program construction. Version 6, August 2004.
5. J.-R. Abrial and L. Mussat. Specification and design of a transmission protocol by successive refinements using B. In M. Broy and B. Schieder, editors, *Mathematical Methods in Program Development*. Springer-Verlag, 1996.
6. R. J. R. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag, 1998.
7. A. Binemann-Zdanovicz, Z. Kramer, P. Schmidt, and B. Thalheim. ASM support for validation and specifications: Lessons learned from an eGovernement project. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
8. A. Blass and Y. Gurevich. Abstract State Machines capture parallel algorithms. *ACM Trans. Computational Logic*, 2002.
9. E. Börger. *Berechenbarkeit, Komplexität, Logik*, chapter Dynamische Algebren und Semantik von Prolog, pages 476–499. Vieweg, 3rd edition, 1992.
10. E. Börger. Evolving Algebras and Parnas tables. In H. Ehrig, F. von Henke, J. Meseguer, and M. Wirsing, editors, *Specification and Semantics*. Dagstuhl Seminar No. 9626, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, July 1996.
11. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
12. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
13. E. Börger. A practice-oriented course on the principles of computation, programming, and system design and analysis. In D. Neville, editor, *Teaching Formal Methods*, volume 3294 of *LNCS*, pages 65–84. Springer-Verlag, 2004.
14. E. Börger. Abstract State Machines: A unifying view of models of computation and of system design frameworks. *Annals of Pure and Applied Logic*, 2004, to appear.
15. E. Börger. Design pattern abstractions and Abstract State Machines. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
16. E. Börger. From finite state machines to virtual machines (Illustrating design patterns and event-B models). In E. Cohors-Fresenborg and I. Schwank, editors, *Präzisionswerkzeug Logik–Gedenkschrift zu Ehren von Dieter Rödding*. Forschungsinstitut für Mathematikdidaktik Osnabrück, 2005. ISBN 3-925386-56-4.
17. E. Börger, A. Cavarra, and E. Riccobene. An ASM semantics for UML activity diagrams. In T. Rus, editor, *Algebraic Methodology and Software Technology, 8th Int. Conf., AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000 Proceedings*, volume 1816 of *Lecture Notes in Computer Science*, pages 293–308. Springer-Verlag, 2000.
18. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 2004.
19. E. Börger and D. Sona. A neural abstract machine. *J. Universal Computer Science*, 7(11):1007–1024, 2001.
20. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

21. E. Börger and R. F. Stärk. Exploiting Abstraction for Specification Reuse. The Java/C# Case Study. In M. Bonsangue, editor, *Formal Methods for Components and Objects: Second International Symposium (FMCO 2003 Leiden)*, volume 3188 of *Lecture Notes in Computer Science (ISBN 3-540-22942-6, ISSN 0302-9743)*, pages 42–76. Springer, 2004. .
22. L. Börger. Using ASMs for modeling biological processes. Personal Communication, 1999.
23. P. L. Brantingham, U. Glässer, B. Kinney, K. Singh, and M. Vajihollahi. Modeling urban crime patterns: Viewing multi-agent systems as Abstract State Machines. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
24. P. Clote. Using stochastic ASMs. e-mail to E. Börger, January 13, 2003.
25. O. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press, 1972.
26. W. P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, Cambridge, 1998.
27. J. Derrick and E. Boiten. *Refinement in Z and Object-Z*. Formal Approaches to Computing and Information Technology. Springer-Verlag, 2001.
28. E. W. Dijkstra. Notes on structured programming. In O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors, *Structured Programming*, pages 1–82. Academic Press, 1972.
29. D. Fahland. Ein Ansatz einer Formalen Semantik der Business Process Execution Language for Web Services mit Abstract State Machines. Master's thesis, Humboldt-Universität zu Berlin, June 2004.
30. D. Fahland and W. Reisig. ASM semantics for BPEL: the negative control flow. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
31. R. Farahbod. Extending and refining an abstract operational semantics of the web services architecture for the business process execution language. Master's thesis, Simon Fraser University, Burnaby, Canada, July 2004.
32. R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract operational semantics of the Business Process Execution Language for web services. Technical Report SFU-CMPT-TR 2004-03, Simon Fraser University School of Computing Science, April 2004.
33. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the business process execution language for web services. In W. Zimmermann and B. Thalheim, editors, *Abstract Sate Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer-Verlag, 2004.
34. J. Fitzgerald and P. G. Larsen. *Modeling Systems. Practical Tool and Techniques in Software Development*. Cambridge University Press, Cambridge, 1998.
35. N. G. Fruja. The correctness of the definite assignment analysis in C#. *Journal of Object Technology*, 3(9):29–52, October 2004.
36. N. G. Fruja. Specification and implementation problems for C#. In B. Thalheim and W. Zimmermann, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2004.
37. N. G. Fruja. A modular design for the Common Language Runtime (CLR) architecture. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
38. Y. Gurevich. A new thesis. *Abstracts, American Mathematical Society*, 6(4):317, August 1985.
39. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
40. Y. Gurevich. Sequential Abstract State Machines capture sequential algorithms. *ACM Trans. Computational Logic*, 1(1):77–111, July 2000.
41. D. Haneberg, H. Grandy, G. Schellhorn, and W. Reif. Verifying security protocols: An ASM approach. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
42. C. Heitmeyer. Using SCR methods to capture, document, and verify computer system requirements. In E. Börger, B. Hörger, D. L. Parnas, and D. Rombach, editors, *Requirements Capture, Documentation, and Validation*. Dagstuhl Seminar No. 99241, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 1999.
43. J. Huggins. ASM web site. `http://www.eecs.umich.edu/gasm/`.
44. H. V. Jula. ASM semantics for C# 2.0. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
45. H. V. Jula and N. G. Fruja. An executable specification of C#. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
46. H. Keller and S. Krüger. *ABAP Objects. Einführung in die SAP Programmierung*. SAP Press and Galileo Press, Bonn, 2nd edition, 2001.
47. E. Kohlbrenner, D. Morris, and B. Morris. The history of virtual machines. Web pages at `http://cne.gmu.edu/itcore/virtualmachine/history.htm`.

48. A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal. Modeling and testing hierarchical GUIs. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
49. D. L. Parnas and J. Madey. Functional documents for computer systems. *Sci. of Comput. Program.*, 25:41–62, 1995.
50. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Germany, 1999.
51. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
52. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 2004.
53. K.-D. Schewe and J. Zhao. ASM ground model and refinement for data warehouses. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*. Université de Paris 12, 2005.
54. R. F. Stärk. Formal specification and verification of the C# thread model. *Theoretical Computer Science*, 2005. To appear.
55. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer-Verlag, 2004.
56. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .
57. M. Vajihollahi. High level specification and validation of the business process execution language for web services. Master's thesis, School of Computing Science at Simon Fraser University, March 2004.
58. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), 1971.
59. J. C. P. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.