Linking Content Definition and Analysis to What the Compiler Can Verify

Egon Börger

Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy
boerger@di.unipi.it

Abstract. In this position paper we outline an approach to relate what the verifying compiler verifies to the definition and analysis (experimental validation and mathematical verification) of the application-content of programs. The underlying Abstract State Machines (ASM) method for high-level system design and analysis bridges the gap between informal requirements and detailed code by combining application-centric *experimentally validatable system modeling* with *mathematically verifiable refinements* of abstract models to compiler-verifiable code.

By definition in [35], the program verifier challenge is focussed on the correctness of programs: software representations of computer-based systems, to-be-compiled by the verifying compiler. As a consequence, "the criterion of correctness is specified by types, assertions and other redundant annotations associated with the code of the program", where "the compiler will work in combination with other program development and testing tools, to achieve any desired degree of confidence in the structural soundness of the system and the total correctness of its more critical components." [35] Compilable code however is the result of two program development activities, which have to be checked too:

- turning the requirements into *ground models*, accurate "blueprints" of the to-be-implemented piece of "real world", which define the application-centric meaning of programs in an abstract and precise form, prior to coding,
- linking ground models to compilable code by a series of *refinements*, which introduce step by step the details *resulting from the design decisions* for the implementation.

We propose to contribute to the program verifier challenge by relating the verification of the correctness for compilable programs to the experimental validation of the application-domain-based semantical correctness for *ground models* and to the mathematical verification of their *refinements* to compilable code, using Abstract State Machine (ASM) ground models [8] and ASM refinements [9].

1 ASM Ground Models (System Blueprints): A Semantical Foundation for Program Verification

Compilable programs, though often considered as the true definition of the system they represent, in many complex applications do however not "ground the

design in reality", since they provide no correspondence between the extra-logical theoretical terms appearing in the code and their empirical interpretation, as requested by a basic principle of Carnap's analysis of scientific theories [20]. By ground models for software systems I mean mathematical application-centric models, which define what Brooks [19] calls "the conceptual construct" or the "essence" of code for a computer-based system and thus "ground the design in reality". Ground models are the result of the notoriously difficult and error prone elicitation of requirements, largely a *formalization* and clarification task realizing the transition from mostly natural-language problem descriptions to a sufficiently precise, unambiguous, consistent, complete and minimal formulation, which represents the algorithmic content of the software contract.

By its epistemological role of relating some piece of "reality" to a linguistic description, the fundamental concept of ground model has no purely mathematical definition, though it can be given a scientific definition in terms of basic epistemological concepts which have been elaborated for empirical sciences by analytic philosophers, see for example [33, 34]. We limit ourselves here to cite from [8] the essential properties which characterize the notion of ground models and can all be satisfied by ASM ground models. Ground models must be:

- *precise* at the appropriate level of detailing yet *flexible*, to satisfy the required accuracy exactly, without adding unnecessary precision;
- *simple and concise* to be understandable by both domain experts and system designers—thus helping designers to resolve the "lack of scientific understanding on the part of their customers (and themselves)" [35, p.66]—and to be manageable for inspection and analysis, avoiding any extraneous encoding and through their abstractions "directly" reflecting the structure of the real-world problem;
- *abstract (minimal) yet complete*. *Completeness* means that every semantically relevant feature is present, that all contract benefits and obligations are mentioned and that there are no hidden clauses. In particular, a ground model must contain as interface all semantically relevant parameters concerning the interaction with the environment, and where appropriate also the basic architectural system structure. The completeness property "forces" the requirements engineer, as much as this is possible, to produce a model which is "closed" modulo some "holes", which are however explicitly delineated, including a statement of the assumptions made for them at the abstract level and to be realized through the detailed specification left for later refinements. Model closure implies that no gap in the understanding of "what to build" is left, that every relevant portion of implicit domain knowledge has been made explicit and that there is no missing requirement—avoiding a typical type of software errors that are hard to detect at the level of compilable code [39, Fact 25]. *Minimality* means that the model abstracts from details that are relevant either only for the further design or only for a portion of the application domain which does not influence the system to be built;
- *validatable* and thus in principle falsifiable by experiment, satisfying the basic Popperian criterion for scientific models [38];

– equipped with a simple yet *precise semantical foundation* as a prerequisite for rigorous analysis and reliable tool support.

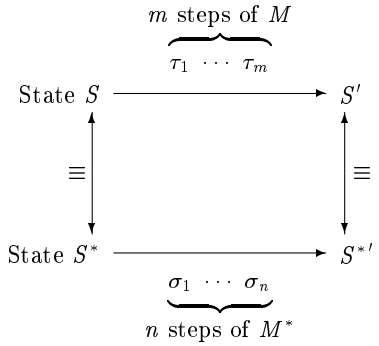
2 ASM Refinements: Management of Design Decisions (Documentation and Verification)

The *ASM refinement notion* I have proposed (for a recent survey see [9]) generalizes Wirth's and Dijkstra's classical refinement method [53, 23]. Using stepwise ASM refinements offers the practitioner a technique to cope with the "explosion of 'derived requirements' (the requirements for a particular design solution) caused by the complexity of the solution process" and encountered "when moving from requirements to design" [39, Fact 26], a process that precedes the definition of compilable code. The ASM refinement method supports practical system validation and verification techniques that split checking complex detailed properties into a series of simpler checks of more abstract properties and their correct refinement, following the path the designer has chosen to rigorously link through various levels of abstraction the system architect's view (at the abstraction level of a blueprint) to the programmer's view (at the level of detail of compilable code). Successive ASM refinements also provide a systematic code development documentation, including behavioral information by state-based abstractions and leading to "further improvements to quality and functionality of the code ... by good documentation of the internal interfaces" [35, p.66].

In choosing how to refine an ASM M to an ASM M^*, one has the freedom to define the following items, as illustrated by Fig. 1:

– a notion (signature and intended meaning) of *refined state*,
– a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, i.e. the pairs of states in the runs one wants to relate through the refinement, including usually the correspondence of initial and (if there are any) of final states,
– a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements),
– a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states,
– a notion of *equivalence* $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to a notion of equivalence of corresponding states of interest.

Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent



With an equivalence notion ≡ between data in locations of interest in corresponding states.

Fig. 1. The ASM refinement scheme

corresponding states. More precisely: fix any notions $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there are an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

The M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$. The states $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) in Fig. 1, for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest). The scheme shows that an ASM refinement allows one to combine in a natural way a change of the signature (through the definition of states and of their correspondence, of corresponding locations and of the equivalence of data) with a change of the control (defining the "flow of operations" appearing in the corresponding computation segments), thus integrating declarative and operational techniques and classical modularization concepts.

The survey in [7] refers to numerous successful practical applications of the above definition, which generalizes other more restricted refinements notions in the literature [40, 41] and scales to the controlled and well documented development of large systems. In particular it supports modularizing ASM refinement correctness proofs aimed at mechanizable proof support, see [40, 48, 13, 17].

3 Summary of Work Done Using the ASM Method

The ASM method to high-level system design and analysis, which is explained in the AsmBook [18], is characterized by the three notions of ASM, ASM ground model and ASM refinement.

ASMs are naturally defined as extension of Finite State Machines [10]: just replace the two fixed FSM locations *in* and *out*, used for reading input and writing output symbols, by any set of readable and/or writable, possibly parameterized, locations $(l, (p_1, \ldots, p_n))$ that may assume values of whatever types. Such sets of updatable locations represent arbitrarily complex abstract memory or states, what logicians call Tarski structures. Otherwise stated, ASMs are FSMs with generalized instructions of form *If Condition Then Updates*, where the FSM-input-event $in = a$ is extended to an arbitrary first-order expression *Condition* and the FSM-output-operation $out := b$ to an arbitrary set *Updates* of assignments $l(t_1, \ldots, t_n) := t$. This definition supports the intuitive understanding of ASMs as pseudo-code operating on abstract data structures.

Using ASMs as precise mathematical form of ground models [8] that are linked to compilable programs by ASM refinements [9], allows one to address the two sides of the software correctness problem in one framework, namely whether the ground model (read: the specification) faithfully reflects the intensions of the requirements and whether the code satisfies the ground model. Furthermore the ASM framework allows one to apply assertion-based techniques to abstract state-based run-time models, thus combining so-called declarative (static logical) and operational (run-time state-based) methods and avoiding the straitjacket of purely axiomatic descriptions. As a consequence, the ASM method supports practical program design and analysis by the following four activities:

- formulate relevant ground model properties ("assertions as specifications in advance of code" [35, p.66]) in traditional mathematical terms, still free from any further burden and restriction that typically derive from additional concerns about a formalization in a specific logic language underlying a proof calculus one may want to use for logical deduction purposes,
- experimentally validate ground model properties by mental or mechanical simulation, performing experiments with the ground model as systematic attempts a) to "falsify" the model in the Popperian sense [38] against the to-be-encoded piece of reality, and b) to "validate" characteristic sets of scenarios, where "testing gives adequate assurance of serviceability" [35, p.69],
- mathematically verify desired ground model properties (e.g. their consistency), using traditional mathematical or (semi-) automated techniques,
- link ground models in a mathematically verifiable way to compilable code via ASM refinements.

In fact the mathematical character of ASMs and ASM runs allows one to use both inspection – for checking the model correctness and completeness with respect to the problem to be solved –[1] and scientific reasoning to analyze model

[1] The need of "inspection" to establish the necessary "evidence" for the correctness of a ground model can be related to Aristotle's observation in the *Analytica Posteriora*,

behavior and properties, using whatever mathematical reasoning means are appropriate, not restricted by the intrinsic limitations of every specific logic calculus. The standard mathematical framework used with ASMs does not limit the verification space, e.g. by Gödel incompleteness or state explosion or similar insufficient-computing-power phenomena.

The proposal to use Abstract State Machines a) as precise mathematical form of ground models and b) for a generalization of Wirth's and Dijkstra's classical refinement method [53, 23]—to a practical framework supporting a systematic separation, structuring and documentation of orthogonal design decisions—goes back to [4–6] where it was used to define what became the ISO standard of Prolog [12]. Since then numerous successful case studies provided ground models for various industrial standards, e.g. for the forthcoming standard of the Business Process Execution Language for Web Services [51], for the ITU-T standard for SDL-2000 [30], for the de facto standard for Java and the Java Virtual Machine [48], the ECMA standard for C# and the .NET CLR [14, 46], the IEEE-VHDL93 standard [15]. The ASM refinement method has been used in numerous ASM-based design and verification projects surveyed in [7].

The ASM method has been linked to a multitude of analysis methods, in terms of both experimental *validation* of models and mathematical *verification* of their properties. The validation (testing) of ASM models can be obtained by their simulation, which corresponds naturally to the notion of ASM run and is supported by numerous tools to mechanically execute ASMs (*ASM Workbench* [21], *AsmGofer* [43], an Asm2C++ compiler [44], C-based *XASM* [3], .NET-executable *AsmL* engine [27], CoreASM Execution Engine [26]). The verification of model properties is possible due to the mathematical character of ASMs, which means precision at the desired level of rigour. As a consequence any justification technique can be used, from proof sketches over traditional or formalized mathematical proofs [47, 37] to tool supported proof checking or interactive or automatic theorem proving, e.g. by model checkers [52, 22, 29], KIV [42] or PVS [24, 28]. As needed for a comprehensive development and analysis environment, various combinations of such verification and validation methods have been supported and have been used also for the correctness analysis of compilers [25, 36] and hardware [50, 49, 45, 32].

For a survey of applications of the ASM method to the design and the analysis of complex computer-based systems and their verified refinement from ground models to compilable code, including industrial system development and reengineering case studies that show the method to scale to large systems, see http://www.eecs.umich.edu/gasm and the AsmBook [18].

4 A Research Challenge and Some Milestones Ahead

The main goal is to define and to provide tool support for hierarchies of mechanically verifiable ASM refinement patterns, which link in a provably correct

that to provide a foundation for a scientific theory no infinite regress is possible and that the first one of every chain of theories has to be justified by "evident" axioms.

way the application-content of systems, as defined by ground models, to to-be-verified compilable programs. This implies an enhancement of the current tool support for the simulation of ASMs and for the verification of their properties.

A *refinement method milestone* consists in defining practical model refinement schemes to turn model properties into software interface assertions comprising behavioral component aspects, to be used where run-time features are crucial for a satisfactory semantically founded correctness notion for code.

A *refinement verification method milestone* is to enhance leading mechanical verification systems by means to prove the correctness of ASM refinement steps. A subgoal consists in linking ASMs to Event-B [1, 2] along the lines of [11], so that the B verification tool set can be exploited to verify model properties and in particular the correctness of refinement steps.

A *refinement validation method milestone* consists in linking the refinement of ground models to ASM execution tools to make the generation and systematic comparison of corresponding test runs of abstract and refined machines possible. In particular relating system and unit level test results should be supported by this enhancement of ASM execution tools.

A *runtime verification method milestone* consists in instrumenting current ASM execution tools to monitor the truth of selected properties at runtime, enabling in particular the exploration of ground models to detect undesired or hidden effects or missing behavior.

A *re-engineering method milestone* is to define methods to extract ground models from legacy code as basis for analysis (and re-implementation where possible), as done for a middle-size industrial case study in [16].

A *compiler verification milestone* is to verify the verifying compiler itself by extending the work of the Verifix [31] project, where ASM ground models were used to describe the underlying real-life machines to run compilers.

5 Concluding Remark

One reviewer asks what the advantages of the ASM method are over other approaches, whether it is "just a difference of notation" or whether there are "fundamental advantages". The *conceptual simplicity* of ASMs as FSMs updating arbitrary locations (read: general states), coupled to the use of *standard algorithmic notation*, constitutes a practical advantage: it makes ASMs understandable for application-domain experts and familiar to every software practitioner, thus supporting the mediation role ground models play for linking in an objectively checkable way informal requirements (read: natural-language descriptions of real-world phenomena) to mathematical models preceding compilable code. A further practical advantage of the ASM method is that it allows designers, programmers, verifiers and testers a) to exploit the abstraction/refinement pair, within one coherent mathematical framework, for a *systematic separation of different concerns* and b) to use any fruitful *combination of whatever precise techniques* are available—whether or not formalized within a specific logic or

programming language or tool—*to define*, experimentally *validate* and mathematically *verify* a series of accurate system models leading to compilable code.

References

1. J.-R. Abrial. Event based sequential program development: application to constructing a pointer program. In *Proc. FME 2003*, pages 51–74. Springer, 2003.
2. J.-R. Abrial. Event driven distributed program construction. Version 6, August 2004.
3. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.
4. E. Börger. A logical operational semantics for full Prolog. Part I: Selection core and control. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'89. 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 36–64. Springer-Verlag, 1990.
5. E. Börger. A logical operational semantics of full Prolog. Part II: Built-in predicates for database manipulation. In B. Rovan, editor, *Mathematical Foundations of Computer Science*, volume 452 of *LNCS*, pages 1–14. Springer-Verlag, 1990.
6. E. Börger. Logic programming: The Evolving Algebra approach. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 391–395, Elsevier, Amsterdam, 1994.
7. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
8. 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.
9. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
10. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *Proc. FroCoS*, volume 3717 of *LNAI*, Vienna (Austria), September 2005. Springer.
11. 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.
12. E. Börger and K. Dässler. Prolog: DIN papers for discussion. ISO/IEC JTCI SC22 WG17 Prolog Standardization Document 58, National Physical Laboratory, Middlesex, England, 1990.
13. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
14. E. Börger, G. Fruja, V. Gervasi, and R. Stärk. A high-level modular definition of the semantics of C#. *Theoretical Computer Science*, 336(2–3):235–284, 2005.
15. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE Computer Society Press.
16. E. Börger, P. Päppinghaus, and J. Schmid. Report on a practical application of ASMs in software design. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 361–366. Springer-Verlag, 2000.

17. E. Börger and D. Rosenzweig. The WAM – definition and compiler correctness. In C. Beierle and L. Plümer, editors, *Logic Programming: Formal Methods and Practical Applications*, volume 11 of *Studies in Computer Science and Artificial Intelligence*, chapter 2, pages 20–90. North-Holland, 1995.
18. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
19. F. P. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, 1987.
20. R. Carnap. The methodological character of theoretical concepts. In H. Feigl and M. Scriven, editors, *Minnesota Studies in the Philosophy of Science*, volume 2, pages 33–76. University of Minnesota Press, 1956.
21. G. Del Castillo. *The ASM Workbench. A Tool Environment for Computer-Aided Analysis and Validation of Abstract State Machine Models.* PhD thesis, Universität Paderborn, Germany, 2001.
22. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *LNCS*, pages 331–346. Springer-Verlag, 2000.
23. 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.
24. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
25. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on ASMs*, pages 50–67. Magdeburg University, 1998.
26. R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05.* Université de Paris 12, 2005.
27. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
28. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *LNCS*, pages 303–322. Springer-Verlag, 2000.
29. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 2003.
30. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
31. W. Goerigk, A. Dold, T. Gaul, G. Goos, A. Heberle, F. W. von Henke, U. Hoffmann, H. Langmaack, H. Pfeifer, H. Ruess, and W. Zimmermann. Compiler correctness and implementation verification: The verifix approach. In P. Fritzson, editor, *Int. Conf. on Compiler Construction, Proc. Poster Session of CC'96*, Linköping, Sweden, 1996. IDA Technical Report LiTH-IDA-R-96-12.
32. A. Habibi. *Framework for System Level Verification: The SystemC Case.* PhD thesis, Concordia University, Montreal, July 2005.
33. A. M. Haeberer and T. S. E. Maibaum. Scientific rigour, an answer to a pragmatic question: a linguistic framework for software engineering. Number 23 in International Conference on Software Engineering, Toronto, 2001.
34. A. M. Haeberer, T. S. E. Maibaum, and M. V. Cengarle. Knowing what requirements specifications specify. Typoscript, 2001.

35. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
36. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, page 415. Springer-Verlag, 2003.
37. S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In *TR 423 CS Dept ETH Zürich*, 2003.
38. K. Popper. *Logik der Forschung. Zur Erkenntnishtoeire der modernen Naturwissenschaft.* Wien, 1935.
39. R.L.Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley, 2003.
40. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
41. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
42. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
43. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer.
44. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.
45. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines.* PhD thesis, University of Ulm, Germany, 2002.
46. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In B. Thalheim and W. Zimmermann, editors, *Abstract State Machines 2004*, Lecture Notes in Computer Science. Springer-Verlag, 2004.
47. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.
48. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .
49. J. Teich, P. Kutter, and R. Weper. Description and simulation of microprocessor instruction sets using ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 266–286. Springer-Verlag, 2000.
50. J. Teich, R. Weper, D. Fischer, and S. Trinkert. A joint architecture/compiler design environment for ASIPs. In *Proc. Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, pages 26–33, San Jose, CA, USA, November 2000. ACM Press.
51. 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.
52. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.
53. N. Wirth. Program development by stepwise refinement. *Commun. ACM*, 14(4), 1971.