The Abstract State Machines Method for Modular Design and Analysis of Programming Languages

Egon Börger

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

Abstract. We survey the use of Abstract State Machines in the area of programming languages, namely to define behavioral properties of programs at source, intermediate and machine levels in a way that is amenable to mathematical and experimental analysis by practitioners, like correctness and completeness of compilers, etc. We illustrate how theorems about such properties can be integrated into a modular development of programming languages and programs, using as example a Java/JVM compilation correctness theorem about defining, interpreting, compiling, and executing Java/JVM code. We show how programming features (read: programming constructs) modularize not only the source programs, but also the program property statements and their proofs.[1]

1 Introduction

The history of what became *The Abstract State Machines Method for Modular Design and Analysis of Programming Languages* starts with simple small examples of Abstract State Machines (ASMs) from where I learnt the concept[2]—which at the time came with various tentative definitions and under various names: 'dynamic structures', 'dynamic algebras', 'evolving algebras'. The examples specified the semantics of Turing and stack machines [101, Sect. 4, 6] and of some tiny Pascal programs [100, Sect. 10, 11]. The idea to use 'dynamic structures' for an operational definition of the semantics of imperative programming constructs was pursued further in Morris' PhD thesis [131]. Although this model covers only the core language features of Modula-2, it nevertheless is of large size, complicated, hard to understand and not scalable, the technical reason being that it is flat and unstructured.

During the winter of 1988/89 I tried to define a precise, execution-oriented ('operational') yet abstract, 'dynamic algebra' model for the dynamic semantics of the Prolog language that would cover the entire language but nevertheless be

[1] This text originates from an invited lecture for the Workshop on Scalable Language Specification at Microsoft Research Cambridge, June 25–27, 2013.

[2] Listening to lectures I had invited Yuri Gurevich to deliver in the years 1986, 1987 and 1991 to the computer science PhD program in Pisa. See [57, Ch.9] or [26] for historical details and references.

of manageable size and reflect the behavior of Prolog programs in a transparent way, to be useful to programmers. After an initial failure the goal was achieved in [19,20,21] by defining the notion of an ASM *ground model* [27,29] and an ASM-tailored *stepwise refinement* [28,29] concept for modeling with ASMs, exploiting the possibilities for abstraction that are inherent in the ASM concept (which became essentially stable in 1995 [102]).

As I will explain in Sect. 2.1 and 2.2 stepwise ASM refinements allowed me to separate orthogonal language features by modules of rule sets (*horizontal refinement*) and to deal with them at different levels of detail (*vertical refinement*)[3], leading from the ground model (which in the case of Prolog became the ISO standard definition of its semantics [114]) to a model of its (in the case of Prolog the WAM [52]) implementation. This provided a transparent, modular structure of small, simple ASM component models which are easy to reuse—not only for the *specification* of models, but also for the *analysis* of their properties by both mathematical proof (verification) and experimental validation (simulation and testing). Such model structuring was supported furthermore by a classification of ASM locations and functions into basic and derived, monitored (or input), output, controlled, and shared ones [24,25].

The early work with ASM ground models and their ASM refinements was influenced by my interest in Prolog and thereby focussed on logic programming systems.[4] It turned out however that the three involved concepts—ASM, ASM ground model and ASM refinement—characterize what became known as *ASM method* for the design and analysis of any software-based system [57]. In particular those concepts support the scalability of ASM models for the major programming paradigms, providing justified-to-be-correct, reusable hierarchies of stepwise refined, structured dynamic semantics models, leading from the source code to a machine code view for full-fledged real-life programming languages. This is illustrated in Sect. 2.3,2.4 for object-oriented programming languages. The hierarchy of ASM models for Java (resp. C#) with its JVM (resp. .NET CLR) implementation exhibits the flexibility the ASM method offers to combine design and verification in a way which supports not only model reuse, but

[3] ASM refinement differs from other refinement concepts in the literature by allowing one to refine in combination both the data structures (which make up the ASM state) and the computation steps (which are described by the ASM rules). More precisely, to relate abstract and refined runs for stating and proving the desired run properties, one can determine five features: a) the targeted refined data structure, b) appropriate pairs of corresponding abstract and refined states of interest one wants to relate, c) segments of abstract and refined computation steps leading from one pair of corresponding states of interest to the next one, d) sets of abstract and refined locations of interest one wants to compare, e) the equivalence properties or whatever comparison one wants to establish, see [28,29] for details. Horizontal resp. vertical ASM refinement includes Java's "extends" resp. "implements".

[4] The early work on specification and analysis of logic programming systems using ASMs, carried out in the period 1988–1994, is surveyed in the Proceedings of the first international ASM workshop which was organized as part of the 13th World Computer Congress, see [23].

a *justifiably correct* programming language product line approach, enhancing feature-driven language composition (as proposed in [11]) by feature verification (statement and proof of properties). In Sect. 3 we review applications of ASMs to parallel and domain specific languages, using as example the hierarchy of models for Occam with its Transputer implementation and for other languages with specific features for the design of distributed, real-time and massively parallel systems (including hardware and instruction set architectures) (Sect.3.1), and last but not least for business process modelling languages (Sect.3.2).[5]

2 Modular reusable ASMs for language semantics

This section describes the role of what I called vertical resp. horizontal ASM refinements for defining structured and reusable models for sequential programming languages. For the illustration we use leading logic and object-oriented programming languages (Prolog, Java, C#) and their implementations.

2.1 Logic programming: Prolog to WAM (vertical refinements)

The (dynamic) semantics problem for programming languages consists in providing a *precise description* of the intended behavior ('meaning') of programs which language users (programmers) and compiler developers can *understand correctly* and *rely upon* in the sense that the description can be used as accurate specification a) for the compiler writer of the freedom available for the implementation and b) for the programmer of the program execution, i.e. that the compiled program behaves in accordance with the description of its intended behavior; such descriptions are what I call a ground model. There are numerous well-developed approaches to mathematically define high-level source program level concepts—e.g. operational, denotational, axiomatic—but methods for *justifiably correct* associations of such concepts to code resulting from their compilation and to runs of this code on virtual or physical machines (verified compilation methods) were in the 1990'ies rarely applied to entire real-life programming languages.

To define the Prolog ground model—for a programming language this is an accurate model (a 'blueprint') which expresses the programmer's view as intended by the language manual or a standardization document—I used *horizontal ASM refinements* to split the model into small components for the core of the logic language [19,22] (with only four ASM rules for the user-defined core of Prolog) and for various groups of so-called built-in predicates to manipulate logic programs [20] and specific data structures [21]. After having presented this model for the dynamic semantics of Prolog I have been challenged by Michael

[5] This paper only collects the evidence for the practicality of the ASM method to engineer software-based (in particular programming language) systems in a justifiably correct and scalable way which intimately links design and analysis all the way from requirements capture to code development. A comparison to numerous other approaches adopted by researchers in the communities of programming languages and formal methods is out-of-scope here and must be left to another occasion.

Hanus to use this model to show the correctness of the Warren Abstract Machine (WAM), a virtual machine with dedicated instructions to efficiently execute compiled Prolog programs. The problem could be solved by a series of *vertical ASM refinements*. In [49,50,52] a chain of 12 proven-to-be-correct refinement steps is defined which links via intermediate ASMs the Prolog ground model ASM (in fact its streamlined version in [51]) to an ASM for its implementation by Warren Abstract Machine code. The guideline for defining the intermediate ASMs were practical concerns, namely to successively introduce orthogonal details of WAM-specific data structures and optimization techniques by modular ASM components in such a way that we could explain the rationale for Warren's design ideas—an important didactical and documentation purpose—and find mathematical (objectively checkable) proofs which show their correctness.

These proofs have been successfully checked by the KIV theorem prover[6] and from that work [164,165,158] a scheme has been extracted for proving the correctness of ASM refinements using generalized forward simulation [159].[7] It is worth remarking that to make the proof feasible for the KIV system, in addition to the 12 refinement steps from [52] one more intermediate ASM model had to be introduced to split two concerns the mathematical proof treats in a single step. We happen to know the time needed for the two efforts, namely 6 person months for defining and mathematically verifying the ASMs for the WAM resp. 12 person months for the KIV specification of those ASM models and the mechanical verification of the proofs (not counting 3 months to study the models and proofs in [52] plus 9 months for the needed further development of the prover and the ASM refinement theory in KIV). Gerhard Schellhorn explained to me some time ago that today this work could be done in less (he estimated half the) time, partly due to increased computer performance. It gives some (really 'anecdotal') indication on the cost difference (measured in person months) to be expected when passing from a traditional mathematical to a machine verification.

The Verifix project [94] of the German Research Council showed—long before Leroy's work [127,128,129]—that this use of ASMs for proving the correctness of compilation schemes scales to verifying the correctness of concrete compilers (implementations in binary) compiling into real-life machine languages [92]. A ground model ASM for the DEC-Alpha processor family has been extracted from the manufacturer's handbook [81]; compiler front-ends [107] and back-ends [182,69,70,82,83,84] have been built based on realistic intermediate languages to prove their correctness, using generic PVS theories developed in [68] to define ASM refinement relations; in [95] ASMs have been used to describe verifying compilers (compilers which verify the correctness of the code they generate).

[6] A few refinement steps have also been verified using Isabelle [146].

[7] This ASM refinement concept has been analyzed further by Schellhorn in [160,162,163] for its use in numerous other KIV verifications of ASMs (the most recent one is a Flash file system verification, see https://swt.informatik.uni-augsburg.de/swt/projects/flash.html); a version tailored for the KIV verification of the Mondex protocol [166,106,98,105] appeared in [161].

2.2 Reusing Prolog/WAM models (horizontal refinements)

Building ASM ground models scales: horizontal ASM refinements allow one to easily extend or adapt a ground model in response to additional or changing requirements. This was what we first learnt from a series of experiments with the major extensions and variants of Prolog and their implementations. They could be realized by horizontal ASM refinements (typically of only some of the involved ASM rules, functions or predicates) in the corresponding Prolog/WAM models to express the additional or changed features. The rest of this section provides details on some of these either purely incremental ASM refinements or ASM refinements which involve rule replacements (non-incremental changes).

Examples of horizontal Prolog/WAM refinements For purely incremental (in logic called conservative) extensions we cite the following five examples:

- *constraint* logic programming:
  - a ground model ASM for Colmerauer's Prolog III could be obtained by simply adding to the unifiability check of the Prolog ground model ASM a solvability test for general constraints [54],
  - for the CLP(R) language and its implementation on the Constrained Logic Arithmetical Machine (CLAM)—a development of IBM at Yorktown Heights—a hierarchy leading from a CLP(R) ground model ASM to its proven to be correct CLAM implementation could be obtained from the Prolog-to-WAM hierarchy by adding rules for solving constraints [53],
- logic programming with *polymorphic types*, developed at IBM by the Protos-L language and its implementation on the Protos Abstract Machine (PAM): it was sufficient to add type constraints and a solvability predicate to the ASM Prolog/WAM models to obtain a refined hierarchy leading from a Protos-L ground model ASM to its proven to be correct PAM implementation model [14,16,15],[8]
- *functional* logic programming: a ground model ASM for the functional logic language Babel and its implementation on the Narrowing Machine [43] could be obtained by simply adding to the backtracking rules of the Prolog models rules for the reduction of functional expressions to normal form,
- *object-oriented* Prolog [136,135]: a ground model ASM could be obtained by enriching the four ASM rules for the user-defined core of Prolog [22] with dedicated rules for object creation and deletion, data encapsulation, inheritance, messages, polymorphism and dynamic binding.

Analogous ASM refinements have been developed in [149] to adapt the Prolog ground model ASM to logic programming languages with *parallelism* (Parlog,

[8] In [13] this construction has even be turned into a general implementation scheme for CLP(X) over an unspecified constraint domain X, namely by designing a generic extension WAM(X) of the Warren Abstract Machine and a corresponding generic compilation scheme of CLP(X) programs to WAM(X) code.

Concurrent Prolog, Guarded Horn Clauses, Pandora, see [46,47,148]) and in [5] to the parallel execution of Prolog on distributed memory (see also [139]).

For non-incremental horizontal ASM refinements, where some Prolog ground model ASM items are replaced by modified ones, we refer to the following variants of logic programming languages:

– *declarative* logic programming language Gödel: a ground model ASM could be defined by abstracting in the Prolog ground model ASM from the deterministic and sequential execution strategy of ISO Prolog [48],
– a ground model for a semi-ring based constraint system (with parallelism) [17] could be obtained by replacing the two Call and Select Rules of [19] by a Reduction Rule which activates a child process simultaneously for each alternative of the current process.

A similar adaptation of the Prolog ground model yields a ground model ASM for a *domain-specific scheduler programming* language (HERA [157, Chap. 3.3]), which is tailored for programming scheduling algorithms for business processes on the basis of given heuristics.

2.3 Object-oriented programming: From Java to JVM

The first application of ASMs to model object-oriented programming constructs appeared in [96] providing a succinct operational description of typical object-oriented features like object creation, overriding, dynamic binding and inheritance in the context of data models. In [169] an ASM rule was added to define cooperative message handling, by describing the run-time search of the most specific cooperation contract in the inheritance hierarchy which implements a message involving several objects on the basis of cooperation contracts. Two projects carried out in Ann Arbor belong here: Blakley's PhD thesis [18], where a (still unstructured) ASM model for a subset of Smalltalk is defined, and the work by Wallace [179], where the ASM refinement method is adopted to extend an earlier ASM model for a subset of C to one for a subset of C++.[9] Zamulin's proposal [180] to explicitly extend ASMs to include objects, also in combination with a type discipline [181], came with an illustration by some examples from the C++ Standard Template Library, but unfortunately has not been pursued further.

In [173] we have applied the ASM method to mathematically analyze the at the time major real-life object-oriented programming language Java together with its virtual machine implementation (JVM). We illustrate in the rest of this section how the (1) definition, (2) mathematical verification and (3) experimental validation of the entire language and its compilation could be obtained by

[9] The method of successive ASM refinements had been adopted in Ann Arbor for the first time in [103] to model the dynamic semantics for a subset of C, structuring an earlier flat version which had been still in the spirit of the early unstructured Modula-2 ASM [131] and had inspired a similar project for Cobol (started in [178], though not continued).

combining ASM ground modeling with horizontal and vertical ASM refinements and moreover that (4) the ASM refinement method supports a verified language product line approach. In Sect. 2.4 we explain how the Java/JVM ground models could be reused to yield a ground model for the ECMA standard of C# and its .NET CLR implementation.

Definition The ground model ASM provides an accurate programmers' understanding of Java defining rigorously and faithfully the intentions of the manual, i.e. of what the programmer can expect from the programs when their compilation to bytecode is executed by the JVM. An essential property of a ground model is that its faithfulness, which relates to a domain-specific natural language description, can be checked and justified by an inspection procedure, though not by mathematically proof due to the fact that the intuitive description one has to start from is not of mathematical rigour. Therefore the Java ground model constructs (must) follow closely the descriptions and examples in the manual, to be successfully checkable as correct.

Since an ASM however is mathematically precise, the ground model typically disambiguates, corrects and completes the descriptions in the manual and makes them coherent, wherever necessary.[10]

To cope with the complexity of the language we have structured the ground model—a language interpreter *JavaIntpr*—by splitting it into orthogonal components, namely interpreters $Java_{Exp_f}$ for expression evaluation and $Java_{Stm_f}$ for statement execution for imperative, static class, object-oriented, exception handling and concurrency (thread) features $f \in \{I, C, O, E, T\}$. Using these interpreters the *JavaIntpr* can be defined in steps by a sequence of horizontal refinements:[11]

$$\begin{array}{lllll}
Java_I \\
Java_I & Java_C \\
Java_I & Java_C & Java_O \\
Java_I & Java_C & Java_O & Java_E \\
Java_I & Java_C & Java_O & Java_E & Java_T
\end{array}$$

where

$$Java_f = Java_{Exp_f} \quad Java_{Stm_f} \text{ for } f \in \{I, C, O, E, T\}$$

Moreover, each of the ASMs in the sequence is a conservative (i.e. purely incremental) extension of the previous one, yielding compositional proof techniques to verify properties of the models, as explained in more detail below.

Mathematical Verification As a characteristic example of a property of interest of Java we have used the ground model ASM in [173, Ch.8] to accurately

[10] In fact the ground model definitions revealed some dark corners and bugs in the official descriptions which in the sequel have been repaired by Sun, see e.g. the list in [173, p.4].

[11] We write here $M \quad N$ for the parallel (not sequential) composition of M and N.

formulate and prove under which conditions Java is type safe.[12] It is worth mentioning that both formulation and proof of this property are in terms of interpreter runs (i.e. of the ASM *JavaIntpr*) and that the major effort was spent to rigorously state the precise meaning of type safety, involving in particular a correct definition of the rules of definite assignment [172][13]—which by the way have also to guarantee that the bytecode generated by a correct Java compiler is not rejected by the bytecode verifier, as we proved in [173, Thm.16.5.2 and 17.1.2], see also [172].

For proofs relating Java programs to bytecode generated for them, also for the JVM a ground model ASM (a bytecode interpreter) had to be defined, wherefor we applied the same horizontal language-driven refinement techniques as described above for Java. This allowed us to rigorously state [173, Ch.14.1] the conditions under which a compiler correctness theorem holds and prove the theorem [173, Ch.14.2], linking runs of Java code to runs of their compilation (on what we called the trustful JVM) to establish that (or better in which sense) two 'corresponding' runs 'yield the same result'. Our correctness proof includes the handling of Java exceptions in the JVM, a feature which considerably complicates the bytecode verification, in the presence of embedded subroutines, class and object initialization and concurrently working threads.[14]

Further additional vertical ASM refinements were introduced to mathematically analyze (formulate and verify) soundness and completeness of bytecode verification for what we called the secure JVM [173, Ch.15-18]. In that machine also loader and preparator components appear; a switching component allowed us to separate the treatment of JVM instructions which affect only the current frame from instructions which manipulate the frame stack (namely upon method call and return, class initialization and exception capture). This modularization supports the use of strong induction hypotheses in proving interpreter properties by nested inductions on ASM runs.

Experimental Validation For testing purposes the high-level Java/JVM models have been refined to executable models in AsmGofer, an extension of TkGofer developed by Joachim Schmid [167] to execute ASMs whose external functions are Haskell definable, a convenient language to program the numerous recursive static functions which appear in Java/JVM. The three AsmGofer executable machines Java-ASM, JVM-ASM and Compiler-ASM can be used in various combinations with the Sun-Compiler and the Sun-JVM: comparing a Java-ASM run of a Java source program with the Java run or with the JVM-ASM or the Sun-JVM run of the bytecode compiled by either the Compiler-ASM or the Sun-Compiler

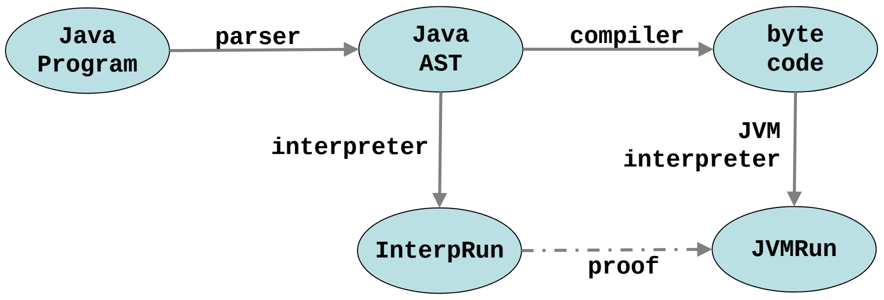
[12] Such a theorem seems not to hold any more for the generic type system of the current version of Java, see [99].

[13] In the Isabelle verification project [118] the rules adopted for definite assignment admittedly omit certain cases which appear in Java.

[14] Note that in the machine verification project reported in [175,119] exception handling has not been taken into account for the correctness proof, though it does appear in the Java sublanguage considered in [118].

(see [173, Appendix A]). Mixing those machines allowed us to isolate whether incoherences we found were bugs in our models or in Sun's specifications and implementations. Contrary to a widely held view we consider it as indispensable for a practical modeling approach that besides the mathematical (verification focussed) model analysis also model execution experiments can be performed.[15]

Verified language product lines It turned out that with ASM refinements not only the models but also the proofs to verify model properties can be reused (modified or extended) for the refined model. In [9] we have shown that the stepwise definition of the Java/JVM models can be linked to a piecemeal formulation and verification of properties of interest, yielding a sort of verified language product line approach. The figure below (taken from op.cit.) illustrates the vertical refinement levels: a grammar G defines Java programs, which are mapped by the parser to an Abstract Syntax Tree, which is used by the Java interpreter *JavaIntpr* for running the Java programs and by the compiler *JavaToJvm* to generate bytecode for them, which is interpreted by the JVM interpreter *JvmIntpr*.



The horizontal ASM refinement steps used in [173] to define these components can be accompanied at each level also by stating and proving properties of interest. In [9] this has been checked in detail for the compiler correctness theorem *Thm* in [173].

For example consider the set Exp_I of imperative Java expressions. The ASMs describing the evaluation of Exp_I expressions come as vertically refined compo-

[15] The railway-related software we developed by the ASM method in a project at Siemens [45] shows an example of an extreme though not rare case where besides an incomplete set of rather loose informal requirements only a set of (here railway transportation) scenarios were given to define the intended behavior of the to be developed software. As a consequence the only thing we could do to calibrate the ground model ASM and to check and justify it in the inspection process as 'correct' was to run the scenarios in the model and check with the experts that the model executes the scenarios as requested. There was no mathematical property around to be proved. The ground model was then compiled to C++ using the compiler developed by J.Schmid [168].

nents for a) the grammar $JavaG_{Exp_I}$ generating Exp_I, b) the Java interpreter $Java_{Exp_I}$ evaluating Exp_I expressions, c) the JVM instruction set $JvmInstr_{Exp_I}$ to which d) the compiler $JavaToJvm_{Exp_I}$ maps Exp_I expressions for execution by e) the JVM interpreter Jvm_{Exp_I} and last but not least f) the compiler correctness theorem Thm_{Exp_I} which is conveniently split into two parts, $ThmS_{Exp_I}$ for the formulation of the statement and $ThmP_{Exp_I}$ for the proof. This yields what following [11] is called a tuple of representations for Exp_I:

$$(JavaG_{Exp_I}, Java_{Exp_I}, JvmInstr_{Exp_I}, Jvm_{Exp_I}, JavaToJvm_{Exp_I}, Thm_{Exp_I})$$

Similar components represent the ASM model concerning the set Stm_I of imperative Java statements. Combining by a horizontal refinement step Exp_I with Stm_I yields the vertical refinement hierarchy of models (a tuple of representations) at the imperative level and can be viewed as defined componentwise by composing the corresponding vertical components (tuple composition $\circ$, we omit the standard grammar components):[16]

$$\begin{aligned}
&(Java_{Stm_I}, Jvm_{Stm_I}, JavaToJvm_{Stm_I}, ThmS_{Stm_I}, ThmP_{Stm_I})\\
&\quad \circ (Java_{Exp_I}, Jvm_{Exp_I}, JavaToJvm_{Exp_I}, ThmS_{Exp_I}, ThmP_{Exp_I})\\
&=\\
&(Java_{Stm_I} \circ Java_{Exp_I}, Jvm_{Stm_I} \circ Jvm_{Exp_I},\\
&\quad JavaToJvm_{Stm_I} \circ JavaToJvm_{Exp_I},\\
&\qquad ThmS_{Stm_I} \circ ThmS_{Exp_I}, ThmP_{Stm_I} \circ ThmP_{Exp_I})
\end{aligned}$$

A purely incremental model refinement corresponds to what in logic is called conservative theory extension so that in this case proving the refined theorem boils down to prove the property in question for the refining features without having to redo the proof for the base model. This has been used extensively in [173]. For example, $Java_{Stm_I}$ adds nine groups of interpreter rules to the six interpreter rule groups of $Java_{Exp_I}$ (one rule group per grammar clause). The compilation component $JavaToJvm_{Stm_I}$ adds eight recursive equations to the equations of $JavaToJvm_{Exp_I}$: six (one per grammar clause) plus eleven equations reflecting the particular compilation of non-strict (Boolean) expressions exploited by the bytecode verfier. $ThmS_{Stm_I}$ adds three invariants (concerning the equivalent positions and computed intermediate values of the two interpreters at the begin resp. (normal or abrupted) end of statement execution) to the five invariants of $ThmS_{Exp_I}$ which are about the equivalence of the values of local Java variables and associated JVM registers resp. about the equivalent positions and computed intermediate values of the two interpreters at the begin/end of an expression evaluation (two invariants for strict and two for non-strict expressions). $ThmP_{Stm_I}$ adds the verification of 22 new cases to the 13 cases verified by $ThmP_{Exp_I}$, where notably $ThmP_{Stm_I}$ uses $ThmP_{Exp_I}$ when invoking the induction hypothesis for expressions occuring in the considered program statement.

[16] This corresponds exactly to the componentwise feature composition in [11], where combining features (read: increments in program functionality) is function composition [10].

The other horizontal refinement levels mentioned above can be dealt with in a similar way, where in some cases some invariants and their proofs are refined (when they are not completely new). See [9, 5.3.,5.4] for the details. Summarizing one sees that the modular character of the ASM models corresponds to a compositional structure of statements and proofs of run properties; within such a structure one can locate extension or change points where more structure can be added. Features exploit such structure variability to link modular design to statement and proof of run invariants.

2.4 Reusing Java ground model for ECMA standard C#

As part of the Microsoft Research ROTOR project we carried out a challenging ASM reuse case study. We tried to reuse as much as possible the various Java/JVM component ASMs and the structure of their combination described above to provide a ground model for (the ECMA standard of) the richer and more complex language C# and its .NET CLR implementation, adopting appropriate modifications or extensions only where imposed by essential differences between the two languages. Not surprisingly the model we obtained [38] clarified a certain number of semantically relevant issues which were not handled by the ECMA standard (but only in the implementation) and detected a series of bugs and gaps in the ECMA standard and in its implementation in .NET as well as some incoherences between the two (see [76] for a detailed account). Also not surprisingly the correctness of the definite assignment analysis could be proved for C# [76] in a way similar to the proof developed for Java. In [171] the C# ground model has been extended by a component for multi-threaded C# and the .NET memory model, the latter one inspired by the ASM developed in [6] for Java's local consistency memory model, and has been used in [170] for a mathematical analysis (using Stärk's AsmTP system, an interactive proof assistant based on ASM logic) of various thread model properties.

As an afterthought we understood that the abstraction potential of ASMs allows one to do still better. A common mathematical structure underlying Java and C# can be made explicit by an ASM which is stepwise refinable to the Java resp. C# ground model. In fact in [58] we have defined hierarchically structured ASM components of an interpreter for a general object-oriented programming language identifying a certain number of static and dynamic parameters that can be refined (in fact instantiated) in two ways to obtain an interpreter for Java resp. C#. The main (in particular semantical) differences between the two languages appear to be localizable by groups of ASM rule sets for clusters of language constructs. This is very closely related to the *incremental semantics* approach in [67] where the semantics of a language is defined by a collection of individual language construct descriptions.

In his PhD thesis Fruja has continued the comparative analysis between the two languages at the implementation level, identifying analogous similarities and differences for the JVM and the .NET CLR [77] models and for mathematical proofs of their properties, in particular for .NET CLR exception handling [80] (reusing parts of the Java/JVM analogue [55]) and .NET CIL type safety [78,79].

3 ASMs for parallel and domain specific languages

In a systematic attempt to test the range of applicability of the ASM method, ASMs have been tried out also for the design and analysis of various domain-specific languages, ranging from languages with generic support for programming parallel, distributed and real-time systems (including hardware and instruction set architectures) (Sect. 3.1) to languages for the specific design of hardware, of business processes and of event-driven database control (Sect. 3.2).

3.1 Parallel and distributed systems programming

Synchronous parallelism is part of the semantics of ASMs since in one step all rules of an ASM are executed simultaneously, a feature which is enhanced by the availability of the **forall** (together with the symmetric **choose**) operator introduced into the final definition of the language in [102]. This directly captures the parallelism needed to model and analyze computer architectures. We showed this by constructing, as part of a reverse engineering project for the massively parallel APE100 architecture [7,8][17], first a programmer's view ground model of the APESE high-level programming language [35] and then its refinement to a register-transfer level model of the control unit processor zCPU [34], a VLSI-implemented microprocessor with pipelining and VLIW parallelism.[18]

The synchronous parallelism in ASMs has been exploited further for the analysis of other pipelining techniques. See for example the series of stepwise refined ASMs in [44] each of which deals with a standard pipelining technique[19], a technique reused in [110] for an advanced commercial RISC processor (though with a simpler pipelining scheme) and in [177] to automatically transform register transfer descriptions of microprocessors into XASM-executable ASMs [1].[20]

In [102] a definition of distributed (in [57] called asynchronous) ASM runs is provided which superseded earlier attempts to introduce true concurrency into the semantics of ASMs.[21] This notion was adopted in all later ASMs that

[17] A rather successful dedicated machine at the time built and used by physicists in Pisa and Rome for floating point intensive numerical simulations in Lattice Gauge Theory.

[18] This instruction set architecture modeling method has been enhanced in [65,64] to instrument models to collect data for evaluating design alternatives.

[19] The first of these refinement steps were checked in [86,174] using the KIV system and PVS, but an omission of a hazard case in the last refinement step remained undetected until Hinrichsen discovered it during his work on generating pipelined systems from sequential processor specifications [108].

[20] Remarkably this allowed one to generate a simulator for a processor architecture from its netlist description or from a graphical description of its data-path, an approach which was pursued in Teich's architecture and compiler co-generation project at the University of Paderborn [176] where ASMs and their XASM [1] executable versions were systematically applied to the hierarchical modeling of application specific instruction set processors.

[21] The ASM model in [104] for the parallelism of Occam, presented in Gurevich's May 1990 lectures in Pisa, inspired the work on modelling the various forms of parallelism

deal with concurrency, including the ones mentioned above to model thread handling in Java/C#. From the point of view of modeling programming languages two early uses of the notion are worth to be mentioned here. In [39,40] an asynchronous ASM is defined as ground model ASM for the Parallel Virtual Machine [85] (PVM, a general-purpose environment for programming heterogeneous distributed processes) at the C-interface level, with an event handling mechanism and message-passing interface which reflect the uniform (multicast or point-to-point) asynchronous access PVM agents (called there “daemons”) have to daemons on other host machines.[22] In [37] the semantics of truly concurrent non-deterministic Occam programs is defined once more, this time however using asynchronous ASMs and proceeding by proven to be correct refinement steps, leading from a programmer level ground model to a processor that runs a high- and a low-priority queue of Occam processes—which we mapped in [36] through a hierarchy of furthermore refined ASMs to an ASM at the Transputer code level. Our guide for these Transputer instruction set architecture refinement steps was the standard Occam-to-Transputer compilation scheme defined by Inmos [112,113] which thereby was proven to be correct.[23]

3.2 ASM interpreters for domain specific languages

ASM models have been used for the design of rather different domain-specific languages. The first example appeared in [157, Chap. 3.3]: the HERA language for programming scheduling algorithms for business processes, obtained as mentioned above by a refinement of the Prolog ground model. In [12] ASMs were used to describe the semantics of a language tailored to program the control for event-driven database applications. In [41,42] we defined an asynchronous ground model ASM for the, at the time new, IEEE standard [111] of the hardware design language VHDL, including the characteristic signal behavior and time model (with pulse rejection limits and the various wait and signal assignment statements involved in the subtle issues related to postponed processes). These ASM models were reused a) in W. Müller’s PhD thesis [137] for defining the semantics of a pictorial extension PHDL of VHDL’93, b) by a group of Toshiba engineers for an extension to analog VHDL and Verilog [155,156,152,154,153], and c) for an adaptation to SystemC [133,134] and to SpecC [132]. Montages[24]

in logic programming languages mentioned above [148,46,47,17] and in the Chemical Abstract Machine and the π-calculus [91].

[22] In [140] modeling PVM is reused to define the semantics of grid systems.

[23] A correctness proof for executing compiled Occam programs [138] on the Transputer architecture [97] had been one of the goals of the European ProCoS project on provably correct systems [125].

[24] Montages [123,4] added to ASMs the possibility to combine graphical and textual elements for the simultaneous specification of the static and dynamic semantics of programming languages, exploiting the syntax-driven modularity in sequential languages. See the complete Montages definition of the syntax, static and dynamic semantics of Oberon in [122] and of C in [109]. The development tool Gem-Mex [3] which allows one to create Montages has been applied in [2] to provide an executable

has been successfully applied to the design (specification and implementation) of a driver specification language needed to solve a complex data warehouse problem at the Union Bank of Switzerland [124]. In [62] an ASM is defined for P3L, a programming language with task and data parallelism, describing the compiler generation of a network of processes and their runs.

An outstanding example is the ground model ASM with an AsmL-executable refinement [71] for SDL2000, a rather expressive language to design distributed real-time (in particular industrial telecommunication) systems. This modeling project has been proposed to the SDL Forum in [87] (after a feasibility test with a ground model ASM for Basic SDL-92 [90]) and then has been carried out in [88,89,72,145]. In 2000 the international standardization body ITU-T for telecommunications adopted the ground model as official definition of the standard [115].

ASM interpreters for BPM and web service languages The ASM method has been applied also to languages that have been developed for 'programming in the large' business processes and web services, where typically also graphical design notations are used. A well-kown example is the UML Activity Diagram language for which (at the time it was version 1.3 [63]) a precise dynamic semantics has been defined by a ground model ASM in [60,32]. In [150] Sarstedt defined an ASM to interpret the considerably richer version 2.0 of (a token flow view [151] of) UML activity diagrams and implemented it by a runtime component which is part of an integrated software development environment where it executes (and visualizes) activity diagrams directly. Kohlmeyer [120] integrated into this work ASM models for other UML behavioral diagrams (see [33,142,116,117,61,143]) "by adding new ASM rules and by modifying appropriate parts of the established ASM specifications" [121, p.217], i.e. by two forms of ASM refinement. The result is a rather practical, rigorous, ground model driven development approach for business process design.

BPMN, the recent OMG standard language[144] for business process design, set out to improve on UML Activity Diagrams. In [56] a ground model ASM has been developed to rationally reconstruct the core behavioral BPMN (version 2.0) features, instead of defining only by a reference implementation the many behaviorally relevant issues the standard left open.[25] Furthermore the ASM model strives for reducing the great number of interdefinable BPMN concepts, leaving

semantics for Mosses' Action Notation. In [93] Montages is characterized as a set of parameterized ASMs.

[25] Already for version 1.0 we had developed an ASM [59] where we formulated our critical remarks and some suggestions how to provide a streamlined (read: minimal), complete and disambiguated (though not a formalized) ASM description of the essential behavioral constructs of the standard. We have sent this work to the convenor and some committee members, but the committee decided that also for version 2.0 'The execution semantics are described informally (textually), and this is based on prior research involving the formalization of execution semantics using mathematical formalisms.'[144, p.425]

it to ASM refinements (in particular instantiations of appropriate parameters) to map these concepts to a minimal set of rigorously defined core constructs.[26] Earlier work [73,147] has built ASM models for basic features of what became BPEL [141], an OASIS standard executable language for programming business process behavior using web services as actions and often used as target language to compile BPMN diagrams into executable code.

To avoid the difficulties and idiosyncrasies of BPMN an alternative tool supported high-level business process design language has been proposed under the name Subject-Oriented Business Process Modeling (S-BPM) [75]. To define the truly concurrent semantics of S-BPM we have developed a language interpreter ASM[27] combining stepwise ASM refinements with a *feature-based* approach—where the meaning of the involved concepts is defined construct by construct. Currently a workflow engine based on this ASM specification and using CoreASM [74] is under development at the University of Linz [126].

4 Conclusion: A challenge

As mentioned above various interactive theorem provers have been used for proving properties of ASM models, in particular the correctness of ASM refinement steps; these were mainly KIV, but also PVS, Isabelle, AsmTP. Tool chains have been developed to support the composition of programming language features, but without considering property verifications, see for example [130]. A challenge we see is to support (the reuse of statements and proofs for) theorem refinements corresponding to ASM refinement steps in such a way that it allows the practitioner to integrate program design and the verification of run-time properties in a feature-based compositional approach to programming language and more generally software development. A first step in this direction appeared in [66]: each of a group of programming language features (namely FeatherweightJava with generics) is verified independently using Coq and the prover is instrumented to generate from these proofs for each feature-based defined language variant a correctness proof by mechanical proof composition.

Acknowledgement I wish to express my thanks to the colleagues who helped with critical comments on preliminary versions of this paper: Sven Apel, Don Batory, Fabian Benduhn,Giuseppe Del Castillo, Albert Fleischmann, Alexander Raschke, Stefan Sarstedt, Gerhard Schellhorn, Thomas Thüm.

[26] For analogous modeling and critical analysis of the so-called Workflow Patterns—which have influenced the definition of the BPMN standard—see [30] where four familiar sequential resp. parallel basic patterns are identified whose ASM refinements yield all the 43 Workflow Patterns en vogue at the time. For a detailed critical evaluation of BPMN, Workflow Patterns and the reference implementation YAWL proposed for them see [31].

[27] The software used to transform the pdf-file for Chapter 12 and the Appendix, generated from latex sources, into a Word document and printer-control-compatible format produced a certain number of partly annoying, partly misleading mistakes in [75]. The interested reader can download the pdf-file for the correct text from [183].

References

1. M. Anlauff. XASM – an extensible, component-based Abstract State Machines language. 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 69–90. Springer-Verlag, 2000.
2. M. Anlauff, S. Chakraborty, P. Kutter, A. Pierantonio, and L. Thiele. Generating an Action Notation environment from Montages descriptions. *Software Tools and Technology Transfer*, 3:431–455, 2001.
3. M. Anlauff, P. Kutter, and A. Pierantonio. Montages/Gem-Mex: a meta visual programming generator. TIK-Report 35, ETH Zürich, Switzerland, 1998.
4. M. Anlauff, P. Kutter, and A. Pierantonio. Enhanced control flow graphs in Montages. In D. Bjoerner and M. Broy, editors, *Proc. Perspectives of System Informatics (PSI'99)*, volume 1755 of *Lecture Notes in Computer Science*, pages 40–53. Springer-Verlag, 1999.
5. L. Araujo. Correctness proof of a distributed implementation of Prolog by means of Abstract State Machines. *J. Universal Computer Science*, 3(5):416–422, 1997.
6. V. Awhad and C. Wallace. A unified formal specification and analysis of the new Java memory models. 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 166–185. Springer-Verlag, 2003.
7. A. Bartoloni et al. A hardware implementation of the APE100 architecture. *Int. J. Mod. Phys.*, C(4):969, 1993.
8. A. Bartoloni et al. The software of the APE100 architecture. *Int. J. Mod. Phys.*, C(4):955, 1993.
9. D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, 14(12):2059–2082, 2008. Extended abstract "Coupling Design and Verification in Software Product Lines" of FoIKS 2008 Keynote in: S. Hartmann and G. Kern-Isberner (Eds): FoIKS 2008 (Proc. of *The Fifth International Symposium on Foundations of Information and Knowledge Systems*), Springer LNCS 4932, p.1–4, 2008.
10. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, October 1992.
11. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE TSE*, June 2004.
12. H. Behrends. *Beschreibung ereignisgesteuerter Aktivitäten in datenbankgestützten Informationssystemen*. PhD thesis, University of Oldenburg, Germany, 1995.
13. C. Beierle. Formal design of an abstract machine for constraint logic programming. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 377–382, Elsevier, Amsterdam, 1994.
14. C. Beierle and E. Börger. Correctness proof for the WAM with types. In E. Börger, G. Jäger, H. Kleine Büning, and M. M. Richter, editors, *Computer Science Logic*, volume 626 of *Lecture Notes in Computer Science*, pages 15–34. Springer-Verlag, 1992.
15. C. Beierle and E. Börger. Refinement of a typed WAM extension by polymorphic order-sorted types. *Formal Aspects of Computing*, 8(5):539–564, 1996.
16. C. Beierle and E. Börger. Specification and correctness proof of a WAM extension with abstract type constraints. *Formal Aspects of Computing*, 8(4):428–462, 1996.
17. S. Bistarelli and E. Riccobene. An operational model for the SCLP language. ILPS Workshop on Tools and Environments for CLP held in Port Jefferson USA, 1997.

18. B. Blakley. *A Smalltalk Evolving Algebra and its Uses.* PhD thesis, University of Michigan, Ann Arbor, Michigan, 1992.
19. 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.
20. 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 *Lecture Notes in Computer Science*, pages 1–14. Springer-Verlag, 1990.
21. E. Börger. A logical operational semantics for full Prolog. Part III: Built-in predicates for files, terms, arithmetic and input-output. In Y. N. Moschovakis, editor, *Logic From Computer Science*, volume 21 of *Berkeley Mathematical Sciences Research Institute Publications*, pages 17–50. Springer-Verlag, 1992.
22. E. Börger. A natural formalization of full Prolog. *Newsletter of the Association for Logic Programming*, 5(1):8–9, 1992.
23. 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.
24. E. Börger. Why use Evolving Algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proc. 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-Verlag, 1995.
25. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
26. 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.
27. 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.
28. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
29. E. Börger. Construction and analysis of ground models and their refinements as a foundation for validating computer based systems. *Formal Aspects of Computing*, 19:225–241, 2007.
30. E. Börger. Modeling workflow patterns from first principles. In C. Parent, K.-D. Schewe, V. Storey, and B. Thalheim, editors, *Conceptual Modeling–ER 2007*, volume 4801 of *Lecture Notes in Computer Science*, pages 1–20. Springer-Verlag, 2007.
31. E. Börger. Approaches to modeling business processes. A critical analysis of BPMN, workflow patterns and YAWL. *J. Software and Systems Modeling*, pages 1–14, 2011. DOI: 10.1007/s10270-011-0214-z. ISSN: 1619-1366 (print version), 1619-1374 (electronic version).
32. 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.

33. E. Börger, A. Cavarra, and E. Riccobene. Modeling the dynamics of UML state machines. 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 223–241. Springer-Verlag, 2000.
34. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering Study). In B. Werner, editor, *Proc. 1st IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145–148, November 1995.
35. E. Börger, G. Del Castillo, P. Glavan, and D. Rosenzweig. Towards a mathematical specification of the APE100 architecture: the APESE model. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 396–401, Elsevier, Amsterdam, 1994.
36. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
37. E. Börger, I. Durdanović, and D. Rosenzweig. Occam: Specification and compiler correctness. Part I: Simple mathematical interpreters. In U. Montanari and E. R. Olderog, editors, *Proc. PROCOMET'94 (IFIP Working Conf. on Programming Concepts, Methods and Calculi)*, pages 489–508. North-Holland, 1994.
38. 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.
39. E. Börger and U. Glässer. A formal specification of the PVM architecture. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 402–409, Elsevier, Amsterdam, 1994.
40. E. Börger and U. Glässer. Modeling and Analysis of Distributed and Reactive Systems using Evolving Algebras. In Y. Gurevich and E. Börger, editors, *Evolving Algebras – Mini-Course, BRICS Technical Report BRICS-NS-95-4*, pages 128–153. University of Aarhus, Denmark, July 1995.
41. 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.
42. E. Börger, U. Glässer, and W. Müller. Formal definition of an abstract VHDL'93 simulator by ea-machines. In C. Delgado Kloos and P. T. Breuer, editors, *Formal Semantics for VHDL*, pages 107–139. Kluwer Academic Publishers, 1995.
43. E. Börger, F. J. López-Fraguas, and M. Rodríguez-Artalejo. A model for mathematical analysis of functional logic programs and their implementations. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 410–415, Elsevier, Amsterdam, 1994.
44. E. Börger and S. Mazzanti. A practical method for rigorously controllable hardware design. In J. P. Bowen, M. B. Hinchey, and D. Till, editors, *ZUM'97: The Z Formal Specification Notation*, volume 1212 of *Lecture Notes in Computer Science*, pages 151–187. Springer-Verlag, 1997.
45. 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 *Lecture Notes in Computer Science*, pages 361–366. Springer-Verlag, 2000.
46. E. Börger and E. Riccobene. A mathematical model of concurrent Prolog. Research Report CSTR-92-15, Dept. of Computer Science, University of Bristol, Bristol, England, 1992.

47. E. Börger and E. Riccobene. A formal specification of Parlog. In M. Droste and Y. Gurevich, editors, *Semantics of Programming Languages and Model Theory*, pages 1–42. Gordon and Breach, 1993.
48. E. Börger and E. Riccobene. Logic + control revisited: An abstract interpreter for Gödel programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 231–154. Oxford University Press, 1994.
49. E. Börger and D. Rosenzweig. From Prolog algebras towards WAM – a mathematical study of implementation. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 31–66. Springer-Verlag, 1991.
50. E. Börger and D. Rosenzweig. WAM algebras – a mathematical study of implementation, Part 2. In A. Voronkov, editor, *Logic Programming*, volume 592 of *Lecture Notes in Artificial Intelligence*, pages 35–54. Springer-Verlag, 1992.
51. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
52. 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.
53. E. Börger and R. Salamone. CLAM specification for provably correct compilation of CLP($\mathcal{R}$) programs. In E. Börger, editor, *Specification and Validation Methods*, pages 97–130. Oxford University Press, 1995.
54. E. Börger and P. Schmitt. A formal operational semantics for languages of type Prolog III. In E. Börger, H. Kleine Büning, M. M. Richter, and W. Schönfeld, editors, *CSL'90, 4th Workshop on Computer Science Logic*, volume 533 of *Lecture Notes in Computer Science*, pages 67–79. Springer-Verlag, 1991.
55. E. Börger and W. Schulte. A practical method for specification and analysis of exception handling: A Java/JVM case study. *IEEE Trans. Software Eng.*, 26(10):872–887, October 2000.
56. E. Börger and O. Sörensen. BPMN core modeling concepts: Inheritance-based execution semantics. In D. Embley and B. Thalheim, editors, *Handbook of Conceptual Modelling. Theory, Practice, and Research Challenges*, pages 287–332. Springer, 2011. ISBN: 978-3-642-15864-3.
57. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
58. 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. .
59. E. Börger and B. Thalheim. A method for verifiable and validatable business process modeling. In E. Börger and A. Cisternino, editors, *Advances in Software Engineering*, volume 5316 of *LNCS*, pages 59–115. Springer-Verlag, 2008.
60. A. Cavarra. *Applying Abstract State Machines to Formalize and Integrate the UML Lightweight Method.* PhD thesis, University of Catania, Sicily, Italy, 2000.
61. A. Cavarra, E. Riccobene, and P. Scandurra. Integrating UML static and dynamic views and formalizing the interaction mechanism of UML state machines. 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 229–243. Springer-Verlag, 2003.

62. A. Cavarra, E. Riccobene, and A. Zavanella. A formal model for the parallel semantics of P3L. In J. Carroll, E. Damiani, H. Haddad, and D. Oppenheim, editors, *Proc. 2000 ACM Sympos. Applied Computing*, volume 2 of *Lecture Notes in Computer Science*, pages 804–812. ACM Press, 2000.
63. R. S. Corporation. Unified modeling language. http://www.rational.com, 1999. version 1.3.
64. G. Del Castillo and W. Hardt. Fast dynamic analysis of complex hardware/software systems based on Abstract State Machine models. In *Proc. 6th Int. Workshop on Hardware/Software Codesign (CODES/CASHE'98) (March 15–18, Seattle, Washington)*, pages 77–81, 1998.
65. G. Del Castillo and W. Hardt. Towards a unified analysis methodology of HW/SW systems based on Abstract State Machines: Modeling of instruction sets. In *"Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen"*, HNI-Verlagsschriftenreihe Vol.36 (ISBN 3-931466-35-3). HNI Paderborn, 1998.
66. B. Delaware, W. Cook, and D. Batory. Product lines of theorems. In *Proc. 2011 ACM international conference on Object oriented programming systems languages and applications*, pages 595–608. ACM, 2011.
67. K. G. Doh and P. D. Mosses. Composing programming languages by combining action-semantics modules. *Science of Computer Programming*, 47(1):3–36, 2003.
68. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
69. 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 Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
70. A. Dold, T. Gaul, and W. Zimmermann. Mechanized verification of compiler back-ends. In *Proc. STTT '98*, 1998. Describes an approach to mechanically prove the correctness of BURS specifications and shows how such a tool can be connected with BURS based back-end generators. The proofs are based on the operational semantics of both source and target system languages specified by means of ASMs. PVS is used to mechanicalyy verify BURS-rules based on formal representations of the languages involved. PVS proof strategies are defined which enable an automatic verification of the rules.
71. R. Eschbach, U. Gässer, R. Gotzhein, M. v. Löwis, and A. Prinz. Formal definition of SDL-2000 – compiling and running SDL specifications as ASM models. *J. Universal Computer Science*, 7(11):1025–1050, 2001.
72. 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 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 242–265. Springer-Verlag, 2000.
73. 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.
74. R. Farahbod et al. *The CoreASM Project*. http://www.coreasm.org.
75. A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, and E. Börger. *Subject-Oriented Business Process Management*. Springer-Verlag, Heidelberg, 2012.
76. N. G. Fruja. The correctness of the definite assignment analysis in C#. *Journal of Object Technology*, 3(9):29–52, October 2004.

77. N. G. Fruja. A modular design for the Common Language Runtime (CLR) architecture. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*, pages 175–200. Université de Paris 12, 2005.
78. N. G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2006.
79. N. G. Fruja. Towards proving type safety of .NET CIL. *Science of Computer Programming*, 72(3):176–219, 2008.
80. N. G. Fruja and E. Börger. Modeling the .NET CLR exception handling mechanism for a mathematical analysis. *J. Object Technology*, 5(3):5–34, 2006.
81. T. Gaul. An Abstract State Machine specification of the DEC-Alpha processor family. Verifix Working Paper Verifix/UKA/4, University of Karlsruhe, Germany, 1995.
82. T. Gaul, A. Heberle, and W. Zimmermann. An ASM specification of the operational semantics of MIS. Verifix Working Paper Verifix/UKA/3, University of Karlsruhe, Germany, 1998.
83. T. Gaul, A. Heberle, W. Zimmermann, and W. Goerigk. Construction of verified software systems with program-checking: An application to compiler back-ends. In A. Pnueli and P. Traverso, editors, *Proc. RTRV '99: Workshop on Runtime Result Verification*, Trento (Italy), 1999.
84. T. Gaul, W. Zimmermann, and W. Goerigk. Practical construction of correct compiler implementations by runtime result verification. In *Proc. SCI'2000*, pages 406–411, Orlando, Florida, 2000. .
85. A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.
86. M. Giese, D. Kempe, and A. Schönegge. KIV zur Verifikation von ASM-Spezifikationen am Beispiel der DLX-Pipelining Architektur. Interner Bericht 16/97, Universität Karlsruhe, Germany, 1997.
87. U. Glässer. ASM semantics of SDL: Concepts, methods, tools. In Y. Lahav, A. Wolisz, J. Fischer, and E. Holz, editors, *Proc. 1st Workshop of the SDL Forum Society on SDL and MSC*, volume 104 (ISSN 0863-095) of *Informatik-Berichte*, pages 271–280. Humboldt-Universität Berlin, 1998.
88. U. Glässer. *Analysis and Validation of Formal Requirement Specifications in Model-Based Engineering of Concurrent Systems.* Habilitationsschrift, University of Paderborn, Germany, 1999.
89. U. Glässer, R. Gotzhein, and A. Prinz. Towards a new formal SDL semantics based on Abstract State Machines. In G. v. Bochmann, R. Dssouli, and Y. Lahav, editors, *SDL'99 – The Next Millenium, Proc. 9th SDL Forum*, pages 171–190. Elsevier Science B.V., 1999.
90. U. Glässer and R. Karges. Abstract State Machine Semantics of SDL. *J. Universal Computer Science*, 3(12):1382–1414, 1997.
91. P. Glavan and D. Rosenzweig. Communicating evolving algebras. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 182–215. Springer-Verlag, 1993.
92. 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.

93. G. Goos, A. Heberle, W. Löwe, and W. Zimmermann. On modular definitions and implementations of programming languages. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, number 87 in TIK-Report, pages 174–208. ETH Zürich, March 2000.
94. G. Goos, H. von Henke, and H. Langmaack. Project verifix. http://www.info.uni-karlsruhe.de/projects.php/id=28&lang=en.
95. G. Goos and W. Zimmermann. Verifying compilers and 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 177–202. Springer-Verlag, 2000.
96. G. Gottlob, G. Kappel, and M. Schrefl. Semantics of object-oriented data models – the evolving algebra approach. In J. W. Schmidt and A. A. Stogny, editors, *Next Generation Information Technology*, volume 504 of *Lecture Notes in Computer Science*, pages 144–160. Springer-Verlag, 1991.
97. I. Graham. *The Transputer Handbook*. Prentice-Hall, 1990.
98. H. Grandy, M. Bischof, G. Schellhorn, W. Reif, and K. Stenzel. Verification of Mondex Electronic Purses with KIV: From a Security Protocol to Verified Code. In *FM 2008: 15th Int. Symposium on Formal Methods*. Springer LNCS 5014, 2008.
99. D. Grunwald, M. Lochau, E. Börger, and U. Goltz. An abstract state machine model for the generic java type system. Technical Report 2010-02, TU Braunschweig, 2010.
100. 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.
101. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bull. EATCS*, 43:264–284, 1991.
102. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
103. Y. Gurevich and J. Huggins. The semantics of the C programming language. In E. Börger, H. Kleine Büning, G. Jäger, S. Martini, and M. M. Richter, editors, *Computer Science Logic*, volume 702 of *Lecture Notes in Computer Science*, pages 274–309. Springer-Verlag, 1993.
104. Y. Gurevich and L. S. Moss. Algebraic operational semantics and Occam. In E. Börger, H. Kleine Büning, and M. M. Richter, editors, *CSL'89, 3rd Workshop on Computer Science Logic*, volume 440 of *Lecture Notes in Computer Science*, pages 176–192. Springer-Verlag, 1990.
105. D. Haneberg, N. Moebius, W. Reif, G. Schellhorn, and K. Stenzel. Mondex: Engineering a provable secure electronic purse. *International Journal of Software and Informatics*, 5(1):159–184, 2011. http://www.ijsi.org.
106. D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Asp. Comput.*, 20(1):41–59, 2008.
107. A. Heberle, T. Gaul, and W. Zimmermann. Construction of verified compiler front-ends with program-checking. In *Proc. PSI '99*, volume LNCS 1755. Springer, 1999.
108. H. Hinrichsen. Formally correct construction of a pipelined DLX architecture. Technical Report TR 98-5-1, Darmstad University of Technology, Dept. of Electrical and Computer Engineering, Germany, 1998.

109. J. Huggins and W. Shen. The static and dynamic semantics of C. Technical Report CPSC-2000-4, Kettering University, Computer Science Program, Flint, Michigan, 2000.
110. J. Huggins and D. Van Campenhout. Specification and verification of pipelining in the ARM2 RISC microprocessor. *ACM Trans. Des. Autom. of Electron. Syst.*, 3(4):563–580, 1998.
111. IEEE Std 1076-1993. *IEEE Standard VHDL Language Reference Manual.* IEEE, New York, USA, 1993.
112. INMOS. *Transputer Instruction Set – A Compiler Writer's Guide.* Prentice-Hall, Englewood Cliffs, NJ, 1988.
113. INMOS. *Transputer Implementation of Occam – Communication Process Architecture.* Prentice-Hall, Englewood Cliffs, NJ, 1989.
114. ISO. Prolog-Part 1: General core. ISO Standard Information Technology–Programming Languages ISO/IEC 13211-1, ISO/ICE, January 1995.
115. ITU-T. SDL formal semantics definition. ITU-T Recommendation Z.100 Annex F http://www.sdl-forum.org, International Telecommunication Union, November 2000.
116. Y. Jin, R. Esser, and J. W. Janneck. Describing the syntax and semantics of UML statecharts in a heterogeneous modeling environment. In *Proc. DIAGRAMS 2002*, pages 320–334, 2002.
117. J. Jürjens. A UML statecharts semantics with message-passing. In *ACM Symposium on Applied Computing*, pages 1009 – 1013. ACM, 2002.
118. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28(4):619–695, 2006.
119. G. Klein and M. Strecker. Verified bytecode verification and type-certifying compilation. *J. of Logic and Algebraic Programming*, 58(1-2):27–60, 2004.
120. J. Kohlmeyer. *Eine formale Semantik für die Verknüpfung von Verhaltensbeschreibungen in der UML 2.* PhD thesis, Universität Ulm (Germany), 2009.
121. J. Kohlmeyer and W. Guttmann. Unifying the semantics of uml 2 state, activity and interaction diagrams. In *Ershov Memorial Conference*, pages 206–217, 2009.
122. P. Kutter and A. Pierantonio. The formal specification of Oberon. *J. Universal Computer Science*, 3(5):443–503, 1997.
123. P. Kutter and A. Pierantonio. Montages: Specifications of realistic programming languages. *J. Universal Computer Science*, 3(5):416–442, 1997.
124. P. Kutter, D. Schweizer, and L. Thiele. Integrating domain specific language design in the software life cycle. In *Proc. Int. Workshop on Current Trends in Applied Formal Methods*, volume 1641 of *Lecture Notes in Computer Science*, pages 196–212. Springer-Verlag, 1998.
125. H. Langmaack. The ProCoS approach to correct systems. *Real-Time Systems*, 13:253–275, 1997.
126. H. Lerchner. Open S-BPM workflow engine. http://www.i2pm.net/open-s-bpm-workflow-engine, 2013.
127. X. Leroy. The compcert verifed compiler, software and commented proof. Version 1.13 of 2013-03-12 at http://compcert.inria.fr/doc/, 2008.
128. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
129. X. Leroy. A formally verified compiler back-end. *J.of Automated Reasoning*, 43(4):363–446, 2009.
130. J. Liebig, R. Daniel, and S. Apel. Feature-oriented language families: a case study. In *Proc. Seventh International Workshop on Variability Modelling of Software-intensive Systems.* ACM, 2013.

131. J. Morris. *Algebraic Operational Semantics and Modula-2.* PhD thesis, University of Michigan, Ann Arbor, Michigan, 1988.
132. W. Mueller, R. Dömer, and A. Gerstlauer. The formal execution semantics of SpecC. Technical Report TR ICS 01-59, Center for Embedded Computer Systems at the University of California at Irvine, 2001.
133. W. Mueller, J. Ruf, D. W. Hoffmann, J. Gerlach, T. Kropf, and W. Rosenstiehl. The simulation semantics of SystemC. In *Proc. Design Automation and Test in Europe (DATE 2001)*, pages 64–70, IEEE CS Press, March 2001.
134. W. Mueller, J. Ruf, and W. Rosenstiel. An ASM-based semantics of systemC simulation. In W. Mueller, J. Ruf, and W. Rosenstiel, editors, *SystemC - Methodologies and Applications*, pages 97–126. Kluwer Academic Publishers, 2003.
135. B. Müller. *Eine objektorientierte Prolog-Erweiterung zur Entwicklung wissensbasierter Systeme.* PhD thesis, University of Oldenburg, Germany, 1994.
136. B. Müller. A semantics for hybrid object-oriented Prolog systems. In B. Pehrson and I. Simon, editors, *IFIP 13th World Computer Congress*, volume I: Technology/Foundations, pages 428–433, Elsevier, Amsterdam, 1994.
137. W. Müller. *Executable Graphics for VHDL-Based Systems Design.* PhD thesis, University of Paderborn, Germany, 1996.
138. M. Müller-Olm. *Modular Compiler Verification. A Refinement-Algebraic Approach Advocating Stepwise Abstraction*, volume 1283 of *Lecture Notes in Computer Science.* Springer-Verlag, 1997.
139. Z. Németh. Definition of a parallel execution model with Abstract State Machines. *Acta Cybernetica*, 15(3):417–455, 2002.
140. Z. Németh and V. Sunderam. A formal framework for defining grid systems. In *Proc. Int. Sympos. on Cluster Computing and the Grid (CCGrid2002)*, pages 202–211, Berlin, 21–24 May 2002. IEEE Computer Society Press.
141. Oasis. Web services business process execution language version 2.0, April 2007. Oasis Standard.
142. I. Ober. More meaningful UML models. In *Proc. TOOLS*, pages 146–157, Sydney, Australia, 20–23 November 2000. IEEE Computer Society Press.
143. I. Ober. An ASM semantics for UML derived from the meta-model and incorporating actions. 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 356–371. Springer-Verlag, 2003.
144. OmgBpmn. Business Process Model and Notation (BPMN). Version 2.0. http://www.omg.org/spec/BPMN/2.0, January 2011. formal/2011-01-03.
145. A. Prinz. *Formal Semantics for SDL. Definition and Implementation.* Habilitationsschrift, Humboldt University of Berlin, Germany, 2000.
146. C. Pusch. Verification of compiler correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, volume 1125 of *Lecture Notes in Computer Science*, pages 347–362. Springer-Verlag, 1996.
147. U. G. R. Farahbod and M. Vajihollahi. An Abstract Machine Architecture for Web Service Based Business Process Management. *International Journal on Business Process Integration and Management*, 1(4):279–291, 2006.
148. E. Riccobene. A formal computational model for PANDORA. Technical Report CSTR-92-16 and ACRC-92-15, University of Bristol, Department of Computer Science, 1992.
149. E. Riccobene. *Modelli Matematici per Linguaggi Logici.* PhD thesis, University of Catania, Sicily, Italy, Academic year 1991/92.

150. S. Sarstedt. *Semantic Foundation and Tool Support for Model-Driven Development with UML 2 Activity Diagrams*. PhD thesis, Universität Ulm, 2006.
151. S. Sarstedt and W. Guttmann. An ASM semantics of token flow in UML 2 activity diagrams. In *Ershov Memorial Conference*, volume 4378 of *LNCS*, pages 349–362. Springer-Verlag, 2007.
152. H. Sasaki. A formal semantics for Verilog-VHDL simulation interoperability by Abstract State Machines. In *Proc. IEEE Conf. DATE'99 (Design, Automation and Test in Europe), ICM Munich, Germany*, pages 353–357, 9–12 March 1999.
153. H. Sasaki. A formal semantics on net delay in Verilog-HDL. In *Proc. Asia Pacific Conf. on Chip Design Languages (APCHDL'99)*, pages 100–106, Fukuoka, Japan, 6–8 October 1999.
154. H. Sasaki. A new dynamic equation scheduling to extend VHDL-AMS. In *Proc. Asia Pacific Conf. on Chip Design Languages (APCHDL'99)*, pages 47–52, Fukuoka, Japan, 6–8 October 1999.
155. H. Sasaki, K. Mizushima, and T. Sasaki. Semantic validation of VHDL-AMS by an Abstract State Machine. In *Proc. BMAS'97 (IEEE/VIUF Int. Workshop on Behavioral Modeling and Simulation)*, pages 61–68, Arlington, VA, 20–21 October 1997.
156. T. Sasaki, H. Sasaki, and K. Mizushima. Semantic analysis of VHDL-ASM by attribute grammar. In *Proc. FDL'98 (Forum on Design Languages), Lausanne, Switzerland*, pages 123–131, 6–10 September 1998.
157. J. Sauer. *Wissensbasiertes Lösen von Ablaufsplanungsproblemen durch explizite Heuristiken*. PhD thesis, Universität Oldenburg, Germany, 1993.
158. G. Schellhorn. *Verifikation abstrakter Zustandsmaschinen*. PhD thesis, Universität Ulm, Germany, 1999.
159. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
160. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
161. G. Schellhorn. ASM refinement preserving invariants. *J. UCS*, 14(12):1929–1948, 2008.
162. G. Schellhorn. Completeness of ASM refinement. *Electr. Notes Theor. Comput. Sci.*, 214:25–49, 2008.
163. G. Schellhorn. Completeness of fair ASM refinement. *Sci. Comput. Program.*, 76(9):756–773, 2011.
164. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
165. G. Schellhorn and W. Ahrendt. The WAM case study: Verifying compiler correctness for Prolog with KIV. In W. Bibel and P. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume III: Applications, pages 165–194. Kluwer Academic Publishers, 1998.
166. G. Schellhorn, H. Grandy, D. Haneberg, N. Moebius, and W. Reif. A Systematic Verification Approach for Mondex Electronic Purses Using ASMs. In J.-R. Abrial and U. Glässer, editors, *Rigorous Methods for Software Construction and Analysis (Börger Festschrift)*, volume 5115 of *LNCS*, pages 93–110. Springer, 2009.
167. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at `http://www.tydo.de/AsmGofer`.
168. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.

169. M. Schrefl and G. Kappel. Cooperation contracts. In T. J. Teorey, editor, *Proc. 10th Int. Conf. on the Entity Relationship Approach (ER'91)*, pages 285–307, San Mateo, California, 23–25 October 1991. Entity Relationship Institute.
170. R. F. Stärk. Formal specification and verification of the C# thread model. *Theoretical Computer Science*, 343:482–508, 2005.
171. 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 *LNCS*, pages 38–60. Springer-Verlag, 2004.
172. R. F. Stärk and J. Schmid. Completeness of a bytecode verifier and a certifying Java-to-JVM compiler. *J. of Automated Reasoning*, 30:323–361, 323-361.
173. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001.
174. M. M. Stegmüller. Formale Verifikation des DLX RISC-Prozessors: Eine Fallstudie basierend auf abstrakten Zustandsmaschinen. Diplom thesis, University of Ulm, Germany, 1998.
175. M. Strecker. Formal verification of a Java compiler in Isabelle. In A. Voronkov, editor, *CADE-18*, volume 2392 of *LNAI*, pages 63–77. Springer, 2002.
176. J. Teich. Project Buildabong at University of Paderborn. `http://www-date.upb.de/RESEARCH/BUILDABONG/buildabong.html`, 2001.
177. 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.
178. M. Vale. The evolving algebra semantics of COBOL. Part I: Programs and control. Technical Report CSE-TR-162-93, EECS Dept., University of Michigan, 1993.
179. C. Wallace. The semantics of the C++ programming language. In E. Börger, editor, *Specification and Validation Methods*, pages 131–164. Oxford University Press, 1995.
180. A. Zamulin. Object-oriented Abstract State Machines. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 1–21. Otto-von-Guericke-Universität Magedeburg, 1998.
181. A. Zamulin. Generic facilities in object-oriented ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines – ASM 2000, Int. Workshop on Abstract State Machines, Monte Verita, Switzerland, Local Proceedings*, number 87 in TIK-Report, pages 426–446. ETH Zürich, March 2000.
182. W. Zimmerman and T. Gaul. On the construction of correct compiler back-ends: An ASM approach. *J. Universal Computer Science*, 3(5):504–567, 1997.
183. The file for the correct text of the appendix of [75] can be downloaded from the following two sources.

`http://www.hanser.de/buch.asp?isbn=978-3-446-42707-5&area=Wirtschaft,`
`http://www.di.unipi.it/~boerger/Papers/SbpmBookAppendix.pdf`

.