Construction and Analysis of Ground Models and their Refinements as a Foundation for Validating Computer Based Systems

Egon Börger

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

Abstract. We explain why for the verified software challenge proposed in [66, 67] to gain practical impact, one needs to include rigorous definitions and analysis, prior to code development and comprising both experimental validation and mathematical verification, of *ground models*, i.e. blueprints that describe the required application-content of programs. This implies the need to link via successive refinements the relevant properties of such high-level models in a traceable and checkable way to code a compiler can verify. We outline the Abstract State Machines (ASM) method, a discipline for reliable system development which allows one to bridge the gap between informal requirements and executable code by combining application-centric *experimentally validatable system modelling* with *mathematically verifiable stepwise detailing* of abstract models to compile-time-verifiable code.

1 Introduction

By its original definition in [66], the then called "verifying compiler grand 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." The extension in [67] to what is now called "verified software grand challenge" has broadened the focus from "a program verifier that would use logical proof to give an automatic check of the correctness of programs submitted to it" (*op.cit.* pg.1) to a vision that forsees the interactive use of "a range of program construction and analysis tools" (*ibidem*) and mentions the role of "reliable capture of requirements, and their faithful encoding as specifications" (pg.3) and of "other formal and informal documentation"(pg.4) besides "assertions internal to the program". Nevertheless it still shares the bias towards annotated program texts, as the objects to-be-verified, and towards solving "many of the problems of programming error that afflict builders and users of software today" (pg.1).

However, as has been pointed out already in [78], programming as the "activity of design and implementation of programmed solutions" for real-world problems is more than the production of annotated program texts. The correctness of a programmed solution expresses a certain relation between the program and "the affairs of the world that it helps to handle" (*op.cit.* pg.256), which means much more than just having a program text that is "free of all errors of certain clearly specified kinds" [67, pg.3]. The programming and verification process, including the inevitable process of program modification during maintenance in response to changing requirements or environmental conditions, need to refer to models ("theories") of the relevant part of the real world and to their properties against which code may be verified. In fact, compilable code for complex systems is the result of two program development activities, which before the final code yield abstract models that have to be checked too as correct:

- constructing *ground models* that capture and fully document the requirements. By ground models we mean accurate "blueprints" of the piece of "real world" one has to implement, in the semiconductor industry also called "golden models" [92, pg.26][1]. They define, as system reference documentation that binds all parties involved for the entire development and maintenance process, the application-centric meaning of the to-be-constructed programs, including their interaction with and their dependence on the system and software environment, in an abstract and precise form, not only prior to coding, but also formulated in terms of the application domain and at a level of detailing that is determined by the application problem and thus higher than that of compilable code
- 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 argue in this paper, which is a reelaboration of the position paper [17], that a practically relevant verified software project has to be grounded-in-reality by relating the verification of the correctness of compilable programs to the

- *experimental validation* of the application-domain-based semantical correctness for *ground models* (Sect. 2) and to the
- *mathematical verification* of their *refinements* to compilable code (Sect. 3).

We also show (Sect. 4) that the Abstract State Machine (ASM) method, coming with an appropriate notion of ground models [24] and a general notion of ASM refinements [25] that scales to systems of industrial size, can establish such a methodical link between problems and their solutions encoded at the end of the development chain into compilable code. This leads us to propose some research tasks for the verified software endeavour (Sect. 5).

[1] We avoid here the traditional term "specification" because of its frequent narrow understanding as system description by a purely declarative characterization of the desired system properties. System models as we intend them here describe structures and their possible evolution due to certain operations, so that the model may refer to (possibly internal) states, events and state changes of the system to-be-built.

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

A main point where the verified software challenge as formulated in [66, 67] must be extended to become of practical interest concerns the apparently implicit assumption that compilable programs constitute the true definition of the system they represent. The assumption expresses a widespread belief. However, reading a definition given by code at best can inform about what the code does, but rarely this conveys also a clear idea about what the code should do, so that in general code alone cannot "ground the design in reality". This holds epistemologically speaking—compilable programs typically provide no correspondence of the kind requested by a basic principle of Carnap's analysis of scientific theories [40] between the extra-logical theoretical terms appearing in the code and their empirical interpretation—but also from a practical viewpoint since a documentation merely by thousands of lines of code or module calls cannot be grasped and controlled reliably a human mind. The negative (also economical) effect of such a lack of an appropriate system documentation is tangible in numerous famous system breakdowns, which keep the ominous "software crisis" alive already for four decades, and in the typical ad-hoc character of the fixes that too often are made without a deep understanding of the system and the real causes for the failure and therefore cannot guarantee that the next breakdown will not occur soon. See also the concern expressed in [11] that a verifying software project should not be focussed "on the analysis of artifacts (programs) rather than on their design and construction" since

> we cannot expect verification tools to inject high reliability into a program that was not designed with reliability in mind from the beginning. We must think about reliability at every point in the software production process. If the starting point for verification is that we are given a program and must attempt to verify it, we are in a losing position because we have so little leverage to affect the design of that program.

Defining what the software for a computer-based system is supposed to do takes place during the requirements engineering phase[2] in which a correct understanding-by-humans of the system-to-be-built has to be achieved, including an understanding of its interaction with the environment where it is intended to operate, and to be documented in a binding manner for the entire lifetime of the system (from development through evaluation to maintenance[3]) and for all its stakeholders. In [78] the knowledge that is acquired here is interpreted as a theory, in the sense of Ryle [84] and Popper [81], in our specific case "a theory of how certain affairs of the world will be handled by, or supported by, a computer program" (*ibidem* pg.255). Brooks [39] speaks about "the conceptual

[2] We adopt the widespread use of this bombastic term to denote requirements elicitation, capture, analysis and documentation.

[3] This implies that at each system change this documentation is updated correspondingly, to stay in sink with the code.

construct" or the "essence" of the software system, whose definition precedes the development of code that is only its machine-managed representation.

We explain in the next two sections why and how what we call "ground models" can represent this "conceptual construct" of software systems as a reference for its implementation by compilable code ("grounding the design in reality") and the code verification, which makes it mandatory to also check that the transformation of ground models into code (read: by stepwise detailing, called refinement) preserves the application-centric ground model correctness, as will be discussed in Sect. 3. We anticipate that this correctness preservation through refinements helps to also solve a problem that is hardly tackled where code is taken as system definition, namely to faithfully reflect changing requirements and to document their provably correct implementation in a transparent way.

2.1 Three Basic Semantical Ground Model Attributes and How to Establish Them during Requirements Capture

In this subsection we characterize the three basic semantical properties every ground model has to possess and describe the three problems every satisfactory framework for ground models has to solve to be appropriate for establishing those properties for a model proposed as ground model for a system.[4]

The notoriously difficult and error-prone elicitation of requirements is largely a knowledge-acquisition and partly a formalization endeavor, to achieve an accurate understanding of the to-be-programmed task and of its formulation. It has to realize the transition from usually natural-language and loose (so-called "informal") problem descriptions to a sufficiently exact (unambiguous in the given context), minimal and concise formulation in a binding document of "precisely what to build" [39]. It is the role of ground models to represent this application-problem-determined system content, which constitutes what is also called the software contract. To establish that a model is indeed a faithful model of the to-be-constructed system, one must find appropriate ways to explain that the model captures the intended requirements in a way that satisfies the following three basic semantical model properties (**CoCoCo-properties**):

– *Consistency.* This refers to internal consistency and to consistency of different system views (model aspects). It must guarantee that possibly conflicting objectives in the original requirements have been resolved.
– *Correctness.* This attribute expresses that each model element reflects the original intentions and that it is correctly conveyed to the system designer. Paraphrasing Naur [78, pg.256], a ground model must enable "to explain, for each part of the model text and for each of its overall structural characteristics, what aspect or activity of the world is matched by it". Thus, correctness is not achieved by simply translating informal requirements from a natural

[4] Some authors consider these ground model properties and the related problems as belonging to the pragmatics of specifications.

language into an accurate mathematical language, but in addition a content-based justification of the semantical appropriateness of this translation is needed (see below the discussion of the verification-method problem).

– *Completeness.* Completeness here means that every semantically relevant feature is present, including the necessary underlying application-domain knowledge, that all contract benefits and obligations are mentioned and that there are no hidden clauses. Paraphrasing Naur [78, pg.256], "for any relevant aspect or activity of the world" the ground model must enable "to state its manner of mapping in the model text". 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 to produce a model which is "closed". Due to the minimality condition for ground models explained in Sect. 2.2, this closure is modulo some "holes", which are however explicitly delineated, including a statement of the assumptions made for them at the abstract level.[5] How such assumptions will be realized depends on the particular case: for external devices it is the role of the devices to guarantee the assumptions, for internal software components the assumptions have to be guaranteed through the detailed specification via subsequent refinements. Model "closure" implies that no gap in the understanding of "what to build" is left, that there is no missing requirement—avoiding a typical type of software errors that are hard to detect at the level of compilable code [83, Fact 25]—and that every relevant portion of implicit domain knowledge has been made explicit—thus protecting the programmers against error-prone situations where they are forced to take decisions that fall into the responsibility of domain experts only.

Every solution of the system development task in question has to share these three basic semantical attributes. For ground models one has to be able to establish them directly, without the possibility to derive them from properties of another model to which the ground model could be related. To establish the CoCoCo-properties for given models *directly* requires a framework to solve the following three fundamental methodological problems concerning communication, verification and validation, for short called **CVV-problems**.

Communication First of all ground models must be apt to mediate between the application domain, where the task originates which is to be accomplished by the system to be built, and the world of models (which ultimately includes code), where the relevant piece of reality has to be represented. This is largely

[5] A frequent case of such "holes" is represented by external technical devices, which interact via sensors and actuators with the software to-be-built to control them. Here the role of ground models is to define the behaviour of the whole system, as it is supposed to happen in the real world; the specification of the software control system can be extracted from the ground model. See [6, 14] for an example.

a *language and communication problem* between the software designers and the domain experts or customers—in a multi-disciplinary project they will come from completely different disciplines and many of them will not have learnt to write or even to read code—the parties who prior to coding have to come to a common understanding of “what to build”, to be explicitly documented in a contract containing a model which can be inspected by the involved parties and binds them—once accepted—for the rest of the system development and maintenance process[6]. The language in which the ground model is formulated must be appropriate to naturally yet accurately express the relevant features of the given application domain, resulting from what is called domain analysis, and to be understood by the two parties. This includes the capability to calibrate the degree of precision of the language to the given problem, so as to support the concentration on domain issues instead of issues of notation. It also means that the modelling language should come with a general (conceptual and application-oriented) data model together with a general function model (for a process-oriented definition of the system dynamics) and a general interface concept to represent system environments (consisting of the system users and of neighboring systems or applications) and state-based system behaviour, including non-determinism and concurrency.

The communication problem is not restricted to the requirements engineering phase and the parties involved there. It also appears where different groups, possibly working at different places, or multiple members of a large group, have to cooperate on the construction of one software system, like engineers, system architects, designers, programmers, testers, maintenance experts, etc. It is crucial for a realistic verified-programs-project to work with an open yet coherent and accurate linguistic framework that is simple and general enough to solve this communication problem, following the example of the language of mathematics and exact sciences where formal expressions (equations, formulae or formal statements), tables, curves, figures, etc., together with natural language text can form a consistent and in the given context sufficiently rigorous unit. A restriction to the language of high-level programming languages does not solve the communication problem, as is well explained in [2], nor does the restriction to a specific logic or formal (algebraic or similar) specification language.

Verification The second problem a ground model framework must be well-suited to solve is a *verification-method problem.* It is of epistemological nature and stems from the fact that there are no mathematical means to prove the correctness of the transition from an informal to a precise description. Every

[6] This meets a crucial practical concern, as is pointed out in [79]: the ground model allows the customers or future system users to comment upon the system before it is built and to evaluate it after its completion for a verification that it meets the requirements, allowing the involved parties also to settle disputes that may occur during or after the construction process and to communicate and decide upon eventual changes (and which party has to pay for them!), when they appear during later development stages.

chain of models, which formalizes given requirements and comes for each model with a mathematical correctness proof with respect to its predecessor, must end with one primary model[7], which can be related to the requirements only in a direct way, trying to reach by inspection some kind of evidence of the desired correspondence between the model and the aspects and activities of the real world the model is supposed to capture. This relates to Aristotle's observation in the *Analytica Posteriora* 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. Such an "evidence" of correctness is what ground model inspection has to provide[8]. Nothing better can be done, due to the fact that the individual knowledge acquired by the human during the ground model construction necessarily transcends what is recorded in the documented product and cannot be encoded in rules, as is well explained in [78, Sect.4,5].

Two kinds of means are needed to establish the CoCoCo-properties for a ground model. To check the completeness property it must be possible to proceed via inspection (review) of ground models by the application-domain expert[9], where inspection is to be taken in its traditional understanding, which is an "informal" activity but does not exclude concrete pragmatic rules about the content (specific goals) and the procedure (management) of the inspection process. But also appropriate forms of domain-specific reasoning, not limited to formal deductions in *a priori* determined logic systems, have to be available together with formal reasoning methods to support the designer in checking the internal consistency of the model, as well as the consistency of different system views. In particular view consistency often is the result of an involved and complex process of resolving conflicting objectives in the original requirements.

We believe that these two complementary forms of ground model verification are crucial for a realistic requirements-capture method, though in practice reasoning-based checking of ground model properties often is of less importance than concept-focussed model inspection (see [101, 60]). Having both forms of ground model verification provides a framework to extend the verified software project to "fail-proof systems" [68], i.e. *reliable* systems that may not come with zero-defect code and may be built from unreliable parts.

Validation The third problem ground models must help to solve is a *validation problem.* It must be possible to perform repeatable experiments where the model

[7] For this reason *ground models* [21] were originally called *primary models* [20, Sect. 3].

[8] Certainly the epistemological status of the underlying concept of evidence has to and can be clarified. See for example Carnap's confirmation theory or the discussion on the role of axioms in science, e.g. in the controversy between Frege, who held a "platonistic" view, and Hilbert, who held a "formalistic" position on the role of axioms for a foundation of mathematical theories, see [12].

[9] Providing a precise ground against which questions can be formulated, ground models support the Socratic method of asking "ignorant questions" [15] to check whether the semantic interpretation of the informal problem description is correctly captured by the mapping to the terms in the mathematical model.

behaviour can be observed under given conditions and validated (see [64]), in particular to run relevant scenarios (use cases), providing a framework for

– systematic attempts to "falsify" the model in the Popperian sense [80] against (in Naur's terms the "theory" of) the to-be-encoded piece of reality,
– runtime verification and analysis.

This empirical criterion also takes into account that computer-based systems are not purely intellectual artefacts but inserted in a real-world environment, which offers itself more to testing methods than to mathematical verification techniques. Furthermore, use cases often are part of the requirements and thus directly reflectable through simulations. In case an entire system is conceived as defined by executable specifications of use cases (see for example [62]), this is captured by the corresponding ground model run segments (simulations). It is an important technical side-effect that simulations also allow one to define – prior to coding – a precise system-acceptance test plan and thus to use a ground model in two roles: (1) as an accurate requirements specification (to be matched by the application-domain expert against the given requirements) and (2) as a test model (to be matched by the tester against executions of the final code), where we consider environmental conditions as part of the requirements.[10] These two roles support the combination of runtime verification with automatic test generation of the type proposed in [10] and in general of model-based testing [100].

2.2 Three Basic Methodological Ground Model Attributes

By the epistemological role of ground models to relate some piece of "reality" to a linguistic description, what constitutes a ground model has no purely mathematical definition, though a scientific characterization of the notion can be given in terms of epistemological concepts which have been elaborated for empirical sciences by analytic philosophers, see for example [81, 84, 58, 59]. Similarly, there can be a) no algorithmic criteria to decide whether a model is a correct ground model, and b) no algorithmic system of rules one could follow to construct appropriate ground models, although it is possible to formulate and learn problem-specific ground model patterns and structuring principles[11]. The problem is that the construction of ground models is an activity of "matching some significant part and aspect of an activity in the real world to the formal symbol manipulation that can be done by a program running on a computer" [78, pg.253], based on knowledge that "transcends that which is recorded in the documented products" (read: ground model), as is well explained in *op.cit.* pg.256.

What one can do is to formulate methodological properties models must possess to serve as ground models for computer-based systems. Such properties,

[10] We do not discuss here the much debated question whether and to which degree a model of the environmental conditions has to be part of the software itself.

[11] The reader who wants to see a simple example illustrating how to construct a ground model out of a loosely described set of requirements may read [31] and compare it with the other formalizations in the book.

together with the semantical CoCoCo-properties and the necessary adequacy to solve the CVV-problems discussed in the previous section, provide a useful guideline for the choice of an appropriate ground model language. In the International Technology Roadmap for Semiconductors, where the word "golden model" is used instead of "ground model", these models are characterized simply as "models of the system's intended behaviour" [92]. This characteristic follows from the CoCoCo-attributes. We add to it the following three more explicit methodological properties we view as distinctive for models to become adequate as ground models.[12] Ground models should be:

– *precise* at the appropriate level of detailing, to satisfy the required accuracy exactly, without adding unnecessary definitions that would limit the flexibility of multiple correct model instantiations. What is a "satisfactory degree of exactness" and an "appropriate level of detailing" depends on the modelling task and the related verification goal, has no general characterization and can be resolved only case by case and on pragmatic grounds. The required precision has to provide however the basis for validating and verifying the semantical CoCoCo-properties and thus requires that the modelling framework itself is equipped with a simple yet *precise semantical foundation*, a prerequisite for rigorous analysis and reliable tool support.
– *minimal* (abstract). 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. Minimality guarantees that the model does not depend on any peculiarity of a possible concrete implementation so that the solution space of the problem to be solved is not unnecessarily restricted.
– *simple* (concise) to be understandable and acceptable as contract by the two parties involved, domain experts and system designers. This property should make ground models manageable for inspection and analysis, help designers to resolve the "lack of scientific understanding on the part of their customers (and themselves)" [66, p.66] and enable experts to "clearly explain why ... systems indeed work correctly" [2], an ability that characterizes the knowledge embodied in the "theory" the professional designer possesses about the given real-world problem and its solution [78, pg.255-256].

Obviously there can be a multiplicity of different ground models for one system, since there are usually many ways to abstract from only implementation-relevant details. Also changing requirements can yield different ground models, see the explanations below. This reflects the intrinsic creativity of defining ground models, an activity which can never be completely mechanized, although one can learn many rules of thumb. In the case of a reengineering project it can also happen that the code is the ground model, from which a high-level model is to be abstracted–maybe to be shown to be at least in part correctly refined by the existing code–before refining the abstract model to the new code.

[12] We cannot emphasize enough that all these properties have a methodologically defensible meaning, although by their epistemological nature for none of them can a mathematical (let alone a formal) definition be given.

2.3 Language Conditions for Defining Ground Models

Unfortunately it is still strongly debated what kind of language is suited to express ground models and which methods are appropriate for their analysis. To satisfy the above described ground model properties and to serve as basis for a practical program verification project, neither can the ground model language be confined to the syntax of some particular logic or specification language nor can the means of analysis be restricted to *a priori* fixed rule-based (*a fortiori* if mechanized) reasoning schemes, contrary to what some formulations in [66, 67] seem to suggest and also contrary to the view hold in [75] that "the "verification problem" is the theorem proving problem, restricted to a computational logic". It would not solve the communication problem since the thorough training in mathematical logic it requires goes beyond the expertise that can reasonably be expected from every software practitioner or domain expert. In addition, the purely declarative, non-executable character which is intrinsic for logical, purely axiomatic system descriptions does not solve the validation problem. In fact, most of the successful formal method tools, e.g. model checkers or theorem provers, are used for the verification of internal properties of accurate models or of refinements which relate accurate models, much more than to formulate ground models and to relate them to the encoded piece of reality; see for example the successful practical applications of the B-method [3, 1].

To understand what is needed, the pragmatic approach of applied mathematical sciences can help, where each time the degree of rigour (read: of formalization or of detailing) used for definitions and proofs is chosen to suit the problem under study. One has to select a framework that supports intuitive, content-oriented, precise modelling and reasoning, the way domain experts use it for high-level process-oriented system requirement descriptions and software practitioners for their work with pseudo-code. The need to be able to tailor ground models to resemble the structure and to reflect the degree of detail of the targeted real-world problem implies for the used language to offer natural expressions of broad-spectrum data and process-control features: the ability to directly speak about arbitrary objects, their properties, their relations with other objects, the operations one can perform on them under specific conditions.

The well-known mathematical concept of structure, made explicit by Tarski, reflects this general concept of not necessarily implementable data types, whereas the computational aspect of changing data values is naturally expressed using dynamic-change expressions (rules) of the form

if *Event* **then** *Actions*

used in describing behavioural processes as well as processes of thought. Rules of this form are omnipresent in scientific descriptions and occur in particular in UML state or activity diagrams, which are built from branching (condition checking) nodes and action nodes. Obviously multiple interpretations of how such rules are applied are in use, but the important thing is that for each use there is a clear definition of the underlying semantical meaning.

For the sake of generality, *Event*s have to be allowed to express any static or dynamic, process-internal or environmental properties or relations among the relevant objects. Whether the used expressions belong to a particular logic is of concern only when one embarks on mechanically simulating runs of abstract models or on formalizing proofs (for mechanical verifications or proof-theoretic studies). *Actions* must be usable to describe any dynamic (typically local) state change using any of the underlying internal or external operations. Not to miss the needed generality and simplicity, it is important *not* to divorce the declarative expression of events (rule guards) from the operational character of state-changing actions.[13]

2.4 Using ASMs for Defining Ground Models

The language of Abstract State Machines (ASMs), which naturally extend Finite State Machines (FSMs) to work over arbitrary structures [26] [14], uses for the definition of the dynamic behavior sets of rules as above, which at each step are executed simultaneously to change the current state into the next state, modulo assumed values for the monitored environment functions. A complete ASM model consists of its rules together with a definition of its signature (the collection of data types, which defines the notion of machine states) and a list of all assumptions made on the environment (the monitored locations), the underlying timing constraints, the data types, the class of exceptions, but also the computing resources, the users, etc.

– *Event*s are instantiated in ASM rules as arbitrary conditions (of whatever underlying signature for internal or external environmental locations). This generalizes the firing conditions $ctl\_state = i \wedge in = a$ of FSM transitions, which require the FSM to be in a particular internal (control) state i upon reading from a particular input location a symbol or word a assumed as provided by the environment.[15]
– *Actions* are instantiated in ASM rules as sets of *Updates* of arbitrary memory locations f, which are allowed to be parameterized by parameters $a_1, \ldots, a_n$ of whatever type. Also the new values are allowed to be of whatever type. This generalizes the effect of FSM instructions, which update exactly two locations yielding an *out*put value (typically a symbol or word over a given alphabet) and a change of the internal state ctl_state.

[13] This is exactly the opposite of the view taken in some purely logico-axiomatic approaches, as advocated for example in [61, p.89] where it is explained that "the most important characteristic of Z, which singles it out from every other formal method, is that it is completely independent of any idea of computation."
[14] The original definition in [56] is driven by a computation-theoretical concern.
[15] In control-state ASMs as defined in [22], the top level system structure is characterized by FSM-reminiscent control states, which are sometimes also called *modes* and represent a particular way of decomposing the set of states into a partioning of subsets of states one can describe and analyze independently of each other.

In ASMs, memory locations and their values are described not by the three FSM locations *in*, *out*, *ctl_state* denoting numbers and symbols, but by arbitrary expressions of arbitrary types, built from arbitrary static and dynamic internal or external operations that are present in the underlying structure, so that the updates take the form and the mathematical meaning of array variable assignments $f(t_1, \ldots, t_n) := t$. This definition supports the intuitive understanding of ASMs as virtual machines executing pseudo-code operating on abstract data structures. For a detailed definition of this semantics of ASMs, including a formalization using inference rules, see the AsmBook [37, Sect.2.4].

Consequently ASMs turn the intuitive concept of executing activities, consisting of whatever actions triggered under whatever conditions by whatever events, into a precise mathematical notion of the desired generality and thus constitute a conceptually simple, rigorous framework for building ground models satisfying the nine properties listed above.

– Using ASMs for ground models solves the language and communication problem due to the simplicity and generality of the language of ASMs, which uses only the fundamental **if then** construct of human thought with a computational interpretation over arbitrary structures that can be easily grasped by everybody who learnt the general language of science, whether domain expert, system designer or programmer. This combination of generality and simplicity makes it to easy to define within the language specific notations where these present an advantage, e.g. notations that directly reflect application-domain-characteristic concepts or operations. For examples of ASM-formulations of specific constructs which characterize some widely used system description languages see [37, Sect.7.1].
– Using ASMs for ground models solves the verification problem since one can apply to an ASM both standard (pseudo-) code inspection – for checking the model correctness and completeness with respect to the problem to be solved – and reasoning to analyze its consistency and other relevant model properties ("assertions as specifications in advance of code" [66, p.66]). Due to the flexibility of the language of ASMs, such properties can be formulated in application-oriented or 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. The ASM framework also allows one to apply assertion-based techniques to abstract state-based run-time models; this combination of so-called declarative or property-based (static logical) and operational (run-time and state-based) methods avoids the straitjacket of purely axiomatic descriptions. In fact to ASMs one can apply whatever reasoning means are appropriate, ranging from intuitive considerations to formalized mechanically checkable proofs within a specific logic calculus. The mathematical framework into which ASMs are embedded does not limit the verification space to check the CoCoCo-properties, neither by Gödel incompleteness nor by state explosion or similar insufficient-computing-power phenomena. It is important for the practical success of the

ASM method that it advocates a systematic separation of concerns, in particular to separate design from verification and within verification different degrees of detailing justification chains.[16]

- The validation problem is solved by the operational pseudo-code character of ASMs, which come with a standard notion of process, computation or run. Simulations of ground models are used in proofs of run-time properties and supported by various tools that make large classes of ASMs prototypically executable, see Sect.4 or the survey [37, Ch.8].
- Using ASMs for modelling allows one to construct models that can be shown to possess the three basic methodological ground model properties. In fact, ASM ground models can be tailored to the required degree of precision and minimality by the possibility the language of ASMs—essentially the language of mathematics—offers to fine-tune the description of the objects of discourse and their transformations (the "data types") to the intrinsic abstraction level of the application, without *a priori* concern about their executable encoding (which remains a matter of further refinements, see Sect. 3). As pointed out above, ASMs have a precise semantical foundation, which covers also non-determinism and concurrent execution and provides a scientific basis for a rigorous analysis and understanding of ASMs as (possibly distributed) pseudo-code. ASM ground models allow one to achieve conciseness mainly by avoiding any extraneous encoding and by reflecting "directly", through the abstractions, the relevant structure discerned in the real-world problem.

Furthermore, the abstract character of ASM ground models can be exploited to guide the user in the *application-domain-driven selection of test cases*, exhibiting in the specification the relevant environment parts and the properties to be checked, showing how to derive (specify or generate) test cases from use cases. The mathematical character of ASMs allows one to also evaluate the test coverage of thus defined test cases. The operational character of ASMs supports defining in abstract run-time terms the expected system effect on samples – the *oracle definition* which can be used for static testing, where the code is inspected and compared to the specification, but also for dynamic testing where the execution results are compared. In particular in this way one can integrate into the ASM method powerful verification techniques for automating the test case generation, like model-checking, SAT solvers and constraint satisfaction algorithms.[17] By appropriately refining the oracle, one can also *specify and implement a comparator* by determining, for runs of the ground model and the final code, what are the states of interest to be related (spied), the locations of interest to be watched, and when their comparison is considered successful (the test equivalence relation). These features for specifying a comparator, using the knowledge about how the oracle is refined, reflect the ingredients of the general notion of ASM refinements we discuss in Sect. 3.

[16] This pragmatic scientific attitude is in contrast with the widely held belief that "the central concepts in software verification are program code and formal proofs" [91], a view that underlies also the program verification project formulation in [66, 67].

[17] For the formulation of concrete research challenges in this direction see [69].

To provide also some experimental credentials for the statement that the language of ASMs is appropriate as ground model language, we provide in Sect. 4 pointers to some successful applications of ASM ground models for the design and analysis of complex systems and conclude this section with pointers to two examples through which the reader may check our claim. An elementary example, illustrating the construction of ASM ground models out of informal requirements, can be found in [31] and concretely compared to solutions using other modeling frameworks presented in the same book [49]. The most advanced publicly accessible real-life case study, which involves all the ground model and refinement capabilities offered by the ASM method, for both design and analysis, is the Jbook [96]. There, ASM ground models are developed for interpreters of Java and of the Java Virtual Machine (JVM), including a bytecode verifier, together with a high-level definition of a Java-to-JVM compiler. This compiler, which is proven to be correct for legal and well-typed programs, is refined to a form of certifying compiler by annotating the instructions issued by the compiler with type information that can be used to prove the typability of the generated code [96, Theorem 16.5.1]. This is a mathematically accurate form of Sun's off-device pre-verification (without inlining subroutines) and guarantees at compile time that the generated code will pass the bytecode verifier. Apparently it still represents a challenge for current computer-based theorem proving systems to mechanize such bytecode verification proofs for bytecode compiled from Java programs and to reuse those proofs for the compilation of C# programs to CLR code (see the corresponding refined ASM ground models constructed in [30, 50]). Encouraging examples for computer-based proofs of Java/JVM-subset properties can be found in [16, 76, 71].

3 ASM Refinements: Management of Design Decisions, their Verification and Documentation

Typically there is more than one step to go from a ground model to compilable code. For example often a ground model—the requirements specification—is first enriched to a design specification, which is then implemented by code. "... the specification of a large system is not a monolithic text but rather a succession of more and more precise mathematical models taking account gradually of the requirements of the future system" [2]. The characteristic phenomenon that occurs during this process, which eventually yields the definition of the compilable code, is known as "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" [83, Fact 26]. The numerous and often orthogonal design decisions taken during this process have to be integrated into the link one has to establish between the ground model analysis and the verification of compilable code.

The question is how to link the ground model through the intermediate models to compilable code in such a way that the code verification by the compiler can be traced back to the validation or verification of the ground model and vice versa. This is the role of the classical refinement method [103, 44]. We have generalized the underlying refinement concept to ASMs (see [25] for a recent survey) with the goal to support *practical* system validation and verification techniques that scale up to complex systems and make changing requirements traceable.

Differently from most refinement concepts in the literature, ASM refinements are not necessarily syntax-directed but may concern different components which are all affected by some common feature, e.g. security. Nevertheless also particular forms of refinement can be defined which are compositional, for example analogues of the syntax-directed refinement notions of the B-method [3]. ASM refinements allow one to split checking complex detailed properties into a series of simpler checks of more abstract properties and their correct refinement, following the path of design decisions 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). In addition, successive ASM refinements provide a systematic code development documentation, which supports design validation (simulation and inspection), reuse and maintenance and includes behavioural information by state-based abstractions, thus leading to "further improvements to quality and functionality of the code ... by good documentation of the internal interfaces" [66, p.66]. The practitioner can use ASM refinements for reasons mentioned already in the previous section for ASM ground models: namely the transition from one to a more refined model, or vice-versa in the case of a reengineering project, can be fine-tuned to the new details one wants to introduce, without being hindered by any notational overhead.

Using a chain of stepwise refined models enhances the designer's activity, in particular when it comes to react to so-called changing requirements. Having stepwise refined models at their disposal enables the designer and the system maintenance expert to exactly localize the "right" level of abstraction where a desired change has to be performed and from where it has to be transferred to the more detailed lower levels. This supports an explicit tracing of requirements changes from the ground model to code, in a particularly simple way when the changes are purely incremental so that they can be captured by conservative model extensions. Purely incremental requirements changes give rise to multiple ground models, each one reflecting one set of requirements. "Freezing" a set of requirements in one model does not prevent changing that set and formalizing it by a refined ground model as more desired details become visible. This is the place where the ideas about "regression verification" proposed in [97] can be used. A good refinement strategy aims in particular at encapsulating orthogonal features, which can be added incrementally to models in different ways. Therefore a sequence of successive changes down to executable code, triggered by changing a particular feature at a specific level of abstraction, does not produce extraneous additional work but is nothing else than introducing all the details which are needed anyway, however in fully documented single steps rather than in one fell swoop. This makes it easier to understand the changed implementation details and to control their effect on the entire system.

As observed in [2], collections of models at different refinement levels can be exploited also for an efficient composition of large systems, namely by selecting the models that are appropriate for the needed software system features, possibly adapting them by refining them further or implementing them in a particular programming language, with tools and theories that suit these features and facilitate their verification along the lines of the model composition. An example from the area of programming languages is found in the incremental development of models for Java and the JVM in orthogonal layers (see [96]), similarly for C# and the .NET CLR (see [30, 51]), supporting instruction-wise descriptions of individual programming constructs one can put together as needed for a language definition; see also [38] where interpreters for Java and C# are derived by instantiating a general scheme for the interpretation of object-oriented language features. Such a component-wise system definition also supports verifiable definitions of the structure and the semantical content of managed code, moving away from the classical static compile-link-run model of language semantics to where meta-programming, generative programming and multistage programming are leading, namely to work with VM-based (interpreted or compiled) managed code and managed execution. Programs are composed and generated from separately definable code patterns, code fragments written in different languages and/or components according to directives expressed through metadata, instantiating a general problem solution to particular cases of the problem.

We now briefly explain the definition of the ASM refinement scheme.

ASM Refinement Scheme 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

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \cdots \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \cdots \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

With an equivalence notion $\equiv$ 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.

This refinement definition taken from [25] generalizes other more restricted refinement notions in the literature, as analysed in [85, 86], 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 [85, 96, 29, 36].

4 Some Work Done Using the ASM Method

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 [103, 44] to a practical framework supporting a systematic separation, structuring and documentation of orthogonal design decisions goes back to [18–20]. It was used there to define what became the ISO standard of Prolog [28]. Since then numerous case studies provided ground models for various industrial standards, e.g. for the forthcoming standard of the Business Process Execution Language for Web Services [82], for the ITU-T standard for SDL-2000 [54], for the de facto standard for Java and the Java Virtual Machine [96], the ECMA standard for C# and the .NET CLR [30, 94, 51], the IEEE-VHDL93 standard [32]. The ASM refinement method [25] has been used in various ASM-based design and verification projects surveyed in [23].

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 the simulation of ASM runs, which is supported by numerous tools to mechanically execute ASMs (*ASM Workbench* [42], *AsmGofer* [88], an Asm2C++ compiler [89], C-based *XASM* [9], .NET-executable *AsmL* engine [48], CoreASM Execution Engine [47]). The verification of ASM properties has been performed using justification techniques ranging from proof sketches [34] over traditional [29, 33] or formalized mathematical proofs [95, 77] to tool supported proof checking or interactive or automatic theorem proving, e.g. by model checkers [102, 43, 53], KIV [87] or PVS [45, 52]. 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 [46, 70] and hardware [99, 98, 90, 57].

For more 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 re-engineering case studies that show the method to scale to large systems, see the AsmWebsite http://www.eecs.umich.edu/gasm and the AsmBook [37].

5 A Research Challenge and Some Milestones Ahead

The main goal we propose is to lift Hoare's challenge from program verification to a discipline of verifiable system development. This implies the development of an integrated tool support for hierarchies of verifiable and validatable model refinement patterns, which link in a provably correct and modular way the application-content of systems, as defined by ground models, to verifiable compilable programs. We think about extensions and enhancements of the currently available model development and analysis tools, targeted at combining in one project the *definition* of abstract models and their stepwise refinements to compilable code with their *simulation* and *verifications* of their properties.

This overall goal splits into various subgoals for the generation, verification and validation of ground models and their refinements.

A *refinement generator milestone* consists in defining practical model refinement schemes (refinement patterns), which capture established programming knowledge, together with justifications of their correctness. Refinement steps that can be automated are to be expected typically in domain-specific applications (see the discussion of refinement generators in [2, 93]). Where the refinement process is highly creative, interactive schemes can still be helpful to guide the refinement process by a combination of user-insight and mechanized tactics. In particular, these refinement schemes should allow the verifier to turn model properties into software interface assertions comprising behavioural component aspects, to be used where state-based run-time features are crucial for a satisfactory semantically founded correctness notion for code.[18] An example for a concrete subgoal of the refinement method milestone are the stateful specifications for Java libraries advocated in [8].

A *refinement verifier milestone* is to enhance current logical or computer-based verification systems by means to prove the correctness of ASM refinement steps, building upon and extending the work surveyed in [37, Ch.9]. A first step could consist in linking ASMs to Event-B [4, 5] along the lines of [27], so that the B verification tool set can be exploited to verify properties of ASMs and in particular the correctness of ASM refinement steps.

A *refinement validator 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 *ground model pattern* milestone consists in collecting patterns of frequently occurring model schemes, raising the level of abstraction at which popular programming and design patterns are defined. Useful ground model patterns are to be expected in domain-specific applications, as became visible for process interaction schemes [13] and web service mediation [7].

A *runtime verifier milestone* consists in instrumenting current high-level model execution tools (e.g. interpreters for ASMs or event-B models or model checkers for TLA+ models, for more see [63]) 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 behaviour.

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 [35].

A *system certification milestone* is to integrate ground model validation and analysis into industrial system certification processes, to formulate the technical

[18] The distinction between *assertions* and specifications (models) and their properties is taken from [2]. Assertions are predicates that describe certain (typically local) properties one wants to be checked at run-time. Model properties can be more general, for example global properties.

content of software reliability for embedded systems. This effort can build upon the use that has been made of ASM ground models to formulate industrial standards mentioned in Sect. 4.

A *verified compiler milestone* is to verify the verifying compiler itself by extending the work of the Verifix [55] project to build proven to be correct compilers for a variety of real-life languages and target machines, where ASM ground models were used to describe the underlying real-life machines to run compilers. For recent work on compiler certification, guaranteeing that the safety properties proved on source code hold for the executable compiled code as well, see [72]. Such work presupposes in particular rigorous models for the semantics of the underlying source and target program languages.

This list is not exhaustive. There are more problems to be solved, in particular on the requirements engineering side, where the major question is how to turn requirements into rigorous ground models. The literature contains promising examples of application-domain specific approaches to be integrated here, e.g. the SCR method [65] or the Requirements State Machine technique defined in [73], which relates process-control systems to methods for checking a set of criteria identifying missing (as well as incorrect or ambiguous) requirements.

6 Concluding Remark

Our proposal to lift Hoare's program verifier challenge [66, 67] from compilable code to the set of all system development products–including in particular the ground models resulting from the requirements capture, all the intermediate refined models resulting from further design and the compilable code–goes in the same direction as Abrial's proposal of a System Construction Database that "can be used not only to store future software components but also, more importantly, their various *abstract, and later refined, mathematical models* ... Specification, and design, and corresponding tools, are put together with implementation and corresponding tools. In this respect, the System Construction Database contains the on-going design history of the software construction." [2]

One reviewer asked 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". We do not claim any exclusivity for the ASM method and instead advocate the use of coherent combinations of whatever rigorous practical methods can be of help to build provably reliable software. In this perspective, 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. Also Event-B [4, 5] programs, a sublanguage of B [3], share this simplicity. In fact they can be defined by a class of basic ASMs [27].

A further practical advantage of the ASM method is that it offers an open but coherent conceptual framework that allows designers, programmers, verifiers and testers

- to exploit the abstraction/refinement pair for a *systematic separation of different concerns*, like requirements capture, specification, design, coding, model and code maintenance and reuse, validation (testing, inspection), verification, etc.,
- 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. This feature is crucial for a wide-spectrum method as well as for an accurate integration of system design with verification tools (see [41]),
- to objectively document the system design and analysis activities of each stakeholder in such a way that newcomers can gain a complete and correct understanding of the system and of its implementation by studying this information.

In conclusion, the ASM method represents a solid basis for a practical verified-programs project that scales to systems of industrial size.

Acknowledgement We thank Peter Sestoft for having directed our attention to Naur's paper [78] and Natarajan Shankar for making us aware that the need for the concept of "ground model" has recently been recognized in the International Technology Roadmap for Semiconductors [92].[19]

References

1. Methodologies and technologies for industrial strength systems engineering. http://www.matisse.qinetiq.com/, 1999. Project number IST-1999-11435.
2. J.-R. Abrial. On constructing large computerized systems (a position paper). In *[74]*.
3. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
4. J.-R. Abrial. Event based sequential program development: application to constructing a pointer program. In *Proc. FME 2003*, pages 51–74. Springer, 2003.
5. J.-R. Abrial. Event driven distributed program construction. Version 6, August 2004.
6. J.-R. Abrial, E. Börger, and H. Langmaack. The steam boiler case study: Competition of formal program specification and development methods. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*, pages 1–12. Springer-Verlag, 1996.
7. M. Altenhofen, E. Börger, A. Friesen, and J. Lemcke. A high-level specification for virtual providers. *International Journal of Business Process Integration and Management*, 4(1), 2006.

[19] To appear in *Formal Aspects of Computing* 2006

8. R. Alur. Trends and challenges in algorithmic software verification. In *[74]*.
9. M. Anlauff and P. Kutter. Xasm Open Source. http://www.xasm.org/, 2001.
10. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. Lowry, C. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Wahington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.
11. T. Ball. The verified software challenge: A call for a holistic approach to reliability. In *[74]*.
12. D. Barnocchi. L'‘Evidenza’ nell'assiomatica aristotelica (contributo all'interpretazione dell'assiomatica aristotelica alla luce della moderna logica matematica). *Proteus*, 5:133–144, 1971.
13. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, volume 3785 of *LNCS*, pages 5–35. Springer, 2005.
14. C. Beierle, E. Börger, I. Durdanović, U. Glässer, and E. Riccobene. Refining abstract machine specifications of the steam boiler control to well documented executable code. In J.-R. Abrial, E. Börger, and H. Langmaack, editors, *Formal Methods for Industrial Applications. Specifying and Programming the Steam-Boiler Control*, number 1165 in LNCS, pages 62–78. Springer, 1996.
15. D. M. Berry. The importance of ignorance in requirements engineering. *J. Systems and Software*, 28(2):179–184, 1995.
16. G. Betarte, E. Gimenez, C. Loiseaux, and B. Chetali. Formavie: Formal modelling and verification of the java card 2.1.1 security architecture. In *Proc. eSmart*, 2002.
17. E. Börger. Linking content definition and analysis to what the compiler can verify. In *[74]*.
18. 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.
19. 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.
20. 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.
21. 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.
22. 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 *LNCS*, pages 1–43. Springer-Verlag, 1999.
23. 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.
24. 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.

25. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
26. 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.
27. 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*. Forschungsinst. für Mathematikdidaktik Osnabrück, 2006. ISBN 3-925386-56-4.
28. 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.
29. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
30. 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.
31. E. Börger, A. Gargantini, and E. Riccobene. Abstract State Machines. A method for system specification and analysis. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, pages 103–119. HERMES Sc. Publ., 2006.
32. 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.
33. 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 *LNCS*, pages 151–187. Springer-Verlag, 1997.
34. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.
35. 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.
36. 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.
37. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
38. 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 *LNCS*, pages 42–76. Springer, 2004. .
39. F. P. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, 1987.
40. 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.
41. L. de Moura, S. Owre, H. Ruess, J. Rushby, and N.Shankar. Integrating verification components. In *[74]*.

42. 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.
43. 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.
44. 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.
45. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
46. 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.
47. R. Farahbod et al. *The CoreASM Project.* http://www.coreasm.org.
48. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.
49. M. Frappier and H. Habrias. *Software Specification Methods: An Overview Using a Case Study.* HERMES Science Publishing, 2006.
50. N. G. Fruja and E. Börger. Analysis of the .NET CLR Exception Handling. In V. Skala and P. Nienaltowski, editors, *3rd International Conference on .NET Technologies, .NET 2005, Pilsen, Czech Republic*, pages 65–75, May–June 2005.
51. N. G. Fruja and E. Börger. Modeling the .NET CLR Exception Handling Mechanism for a Mathematical Analysis. *Journal of Object Technology*, 5(3):5–34, 2006.
52. 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.
53. 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.
54. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of sdl-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
55. 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.
56. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bull. EATCS*, 43:264–284, 1991.
57. A. Habibi. *Framework for System Level Verification: The SystemC Case.* PhD thesis, Concordia University, Montreal, July 2005.
58. A. M. Haeberer and T. S. E. Maibaum. Scientific rigour, an answer to a pragmatic question: a linguistic framework for software engineering. Number 21 in Proc. International Conference on Software Engineering (ICSE 21), pages 463–472, Toronto, 2001. IEEE CS Press.
59. A. M. Haeberer, T. S. E. Maibaum, and M. V. Cengarle. Knowing what requirements specifications specify. Typoscript, 2001.
60. A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, Sept 1990.

61. J. A. Hall. Taking Z seriously. In *ZUM'97*, volume 1212 of *Lecture Notes in Computer Science*, pages 89–91. Springer-Verlag, 1997.
62. D. Harel and R. Marelly. Capturing and executing behavioral requirements: the play-in/play-out approach. Technical Report MCS01-15, Weizmann Institute of Science, Israel, 2001.
63. K. Havelund and A. Goldberg. Verify your runs. In *[74]*.
64. M. P. E. Heimdahl. Let's not forget validation. In *[74]*.
65. C. Heitmeyer. Using SCR methods to capture, document, and verify computer system requirements. In E. Börger, B. Hörger, D. L. Parnas, and D. Rombach, editors, *Requirements Capture, Documentation, and Validation.* Dagstuhl Seminar No. 99241, Schloss Dagstuhl, 1999.
66. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
67. T. Hoare and J. Misra. Verified software: theories, tools, experiments. Vision of a Grand Challenge project. In *[74]*.
68. G. J. Holzmann and R. Joshi. Reliable software systems design: Defect prevention, detection, and containment. In *[74]*.
69. J.Rushby. Automated test generation and verified software. In *[74]*.
70. 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 *LNCS*, page 415. Springer-Verlag, 2003.
71. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 2006.
72. X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Proc. POPL'06.* ACM, January 2006.
73. N. G. Leveson. Completeness in formal specification language design for process-control systems. In *Formal Methods in Software Practice*, pages 75–87. ACM Press, 2000.
74. B. Meyer, editor. *Proc.IFIP WG Conference on Verified Software: Tools, Techniques, and Experiments*, http://vstte.ethz.ch/papers.html, Zürich, October 2005. ETH.
75. J. S. Moore. A mechanized program verifier. In *[74]*.
76. J. S. Moore. Proving theorems about Java and the JVM with ACL2. In M. Broy and M. Pizka, editors, *Models, Algebras and Logic of Engineering Software*, volume 191. IOS Press, 2003.
77. S. Nanchen and R. F. Stärk. A security logic for Abstract State Machines. In *TR 423 CS Dept ETH Zürich*, 2003.
78. P. Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15:253–261, 1985.
79. D. L. Parnas. The use of precise documentation in software develpment. Tutorial at FM 2006, see http://fm06.mcmaster.ca/t8.htm, August 2006.
80. K. R. Popper. *Logik der Forschung. Zur Erkenntnistheorie der modernen Naturwissenschaft.* (Engl. Translation: The Logic of Scientific Discovery, Hutchinson 1959, Routledge 1992 and 2002), Wien 1935.
81. K. R. Popper and J. C. Eccles. *The Self and its Brain.* Routledge and Kegan Paul, London, 1977.
82. 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*, 2006.

83. R.L.Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley, 2003.
84. R. Ryle. *The Concept of Mind.* Penguin Books, Harmondworth (England), 1963.
85. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
86. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
87. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
88. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer.
89. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.
90. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines.* PhD thesis, University of Ulm, Germany, 2002.
91. C. Schürmann. Meta-logical frameworks and formal digital libraries. In *[74]*.
92. Semiconductor Industry Assoc. International technologoy roadmap for semiconductors. Design. http://www.itrs.net/Common/2005ITRS/Design2005.pdf, 2005.
93. D. R. Smith. Generating programs plus proofs by refinement. In *[74]*.
94. 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.
95. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.
96. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .
97. O. Strichmann and B. Godlin. Regression verification–a practical way to verify programs. In *[74]*.
98. 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.
99. 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.
100. M. Utting. Model-based testing. In *[74]*.
101. J. M. Wing. A specifier's introduction to formal methods. *Computer*, pages 8–24, Sept. 1990.
102. K. Winter. Model checking for Abstract State Machines. *J. Universal Computer Science*, 3(5):689–701, 1997.
103. N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14(4), 1971.