Why Programming Must Be Supported by Modeling and How?

Egon Börger

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

Abstract. The development of code for software intensive systems involves numerous levels of abstraction, leading from requirements to code. Having abstract modeling concepts available as high-level programming constructs helps to define the code and to make sure that when the system runs with the software executed by machines, the software components behave the expected way. We explain in this paper that nevertheless, there remains a gap, which cannot be closed by mere programming methods, but which can be closed if programming is supported by an appropriate modeling framework (a design and analysis method and a language).

1 Introduction

In this paper we use the term programming as short hand for *programming complex systems* where reliability is a concern. So we speak about the development of code for complex software systems or for software intensive systems where it is critical that the software components do what they are supposed to do. Software intensive systems comprise systems where the software and the machines which execute it are only a part of the overall system, where for the code executing computer(s) the other parts appear as environment—technical equipment, physical surrounding, information systems, communication devices, external actors, humans—upon which the behavior of the software components depends and which they affect.

It is characteristic for the (non agile)[1] development of code for software (intensive) systems to involve descriptions at numerous levels of abstraction, leading from requirements through high-level design to machine executable code. For such descriptions, besides natural language (which is normally used to describe the requirements) a huge variety of dedicated languages and frameworks is available, covering the wide spectrum from direct coding in a programming language to pictorial (as such possibly not executable, 'abstract') models, consisting essentially of visual (graphical) descriptions which are then transformed into more detailed textually described models and finally executable code. Well

[1] Without loss of generality we restrict our attention here to non-agile approaches to software development. In fact, it will become clear below why coding alone, as advocated by agile methods, cannot solve what we call the ground model problem.

known examples of such graphical language constructs can be found for example in the OMG-languages UML, BPMN, SysML. In between one finds a myriad of textual, logic-based, formal specification languages, but also domain-specific languages and the interesting development of programming languages which directly offer constructs to express frequently occurring abstract modeling concepts. High-level programming constructs, like data types, collections, etc. clearly help to define the desired code and to justify that when the system runs, with the code executed by computer(s), the software components in fact behave the expected way. One finds them, to mention a few examples, in Java and C#, Scala (https://www.scala-lang.org/), Kotlin (https://kotlinlang.org/), K (http://www.theklanguage.com), etc.

However, natural language cannnot be avoided because it is the 'mother tongue' in which human stakeholders communicate to ensure a common understanding to capture the requirements correctly and completely. In particular, there remains a gap between requirements and code which for reasons of principle cannot be closed by mere programming constructs or model transformations. It appears prominently at the beginning of the chain which links the understanding by application domain experts (customers, users) of the system to-be-built to the behavior of the system when it runs under the control of the software. The question is how to relate in a controllably reliable way real-world items and behavior (objects, events and actions) to corresponding items in a textual or graphical description, whether directly by code or by an abstract model that is transformed in a correctness preserving manner to code.

In this paper we explain the three main facets of this epistemological problem:

- a *communication problem*,
- an *evidence problem*,
- an *experimental validation problem*.

We characterize the intrinsic properties of a language one needs to solve this problem. First of all, such a language must be **understandable** by the main parties involved, namely application domain experts (users, customers, who are not necessarily engineers) and software developers (designers and programmers). They need it as vehicle to intellectually grasp and transmit an adequate comprehension of complex systems. Second, to permit unambiguous system descriptions which are on the one side abstract enough and on the other side accurate enough, the language must allow the stakeholders to **calibrate the degree of precision** of descriptions (read: their level of abstraction) to the given problem and its application domain. Last but not least, the language must allow the software engineers to **link descriptions at different levels of abstraction**— transform models—in a controllably correct and well documented way to code, using a practical refinement method that is supported by techniques for both, experimental validation and mathematical verification (whether informal, rigorous or formal and machine supported).

In Sect. 2 we explain the special status of *ground models*—the 'blueprints' through which domain experts and software developers must reach a common understanding of 'what to build' [23, p.14]—and what this status implies for

their validation as basis for the verifiability of the system, once it is built.[2] In Sect. 3 we explain how and in which sense the intended behavior, once defined by the ground model, can be 'preserved' by the software engineer, in an objectively controllable and well documented way, when building the software components for the system. This preservation of 'behavioral correctness' can be obtained via a chain of successive refinement steps, which piecemeal implement abstract modeling terms by code, providing the details of the 'how to build', and are closely accompanied by corresponding validation and verification steps. It is here that high-level programming languages which offer support for frequently occuring modeling concepts are extremly helpful. We conclude in Sect. 4 with an explanation of the challenging problem to define practical, tool supportable patterns for stepwise refinement coming with compositional verification techniques.

2 Ground Models

The development of reliable software, whose execution does—and can be explained to do—what it is supposed to do, needs a correct understanding and formulation of the project's real-word problem, including the context where the code executing computer is one among multiple components which together are expected to realize the desired overall system behavior. Such a problem description contains three parts:

- The domain experts (customers) are responsible for the **requirements**, which have to be turned into a sufficiently *precise and complete description of the intended system behavior* ('what to build' [23, p.14]), at the level of abstraction and rigor of the given application domain.
- Given the requirements document, the design engineers must distill a **software specification**, that is a sufficiently precise abstract description of the expected behavior of the software when executed by a computer in the system environment.
- Furthermore, the domain experts must provide a complete description of those **domain assumptions**, on the structure and behavior of the system components, on which the system designers can rely when it comes to guarantee that the specification (and later also its implementation) behaves as system component the way the requirements demand to solve the given problem (*correctness property*). This document must make the underlying portion of implicit application-domain knowledge explicit which the software experts need to understand to make sure their understanding of the ground model is correct.

These three documents constitute what I call a *ground model*, although sometimes the term is also used to refer only to the software specification, which is

[2] To avoid a misunderstanding of the term 'ground' in 'ground model', we point out already here that the ground models may and typically do change during the development *process*, but in each development phase there is one well-defined ground model which has to be related to its implementing code.

what will be turned into the code to-be-built. The code development can start from these three documents; it also *must* start from them, if only because it is errorprone and expensive to start coding before all the needed requirements are correctly formulated and put into an inspectable document. But be aware that the ground model is not only an initial model,[3] as used for example in the B-method [1]; it serves a heterogeneous group of stakeholders as the most abstract complete and correct model of the system to-be-built. As a consequence, the ground model is subject to change during the code development process; see the caveat below which explains why the three documents can be considered as finalized only at the end of the project, when the code is defined.

Whereas the system requirements and the domain assumptions are primarily under the responsibility of the domain experts, the software specification is shared by the two parties, standing between the application-domain-focussed requirements with the related domain assumptions and the software which has to be developed.

In what consists the special character of a ground model? Its constituents are targeted to support a common, objectively checkable understanding by humans—experts of different fields—of some desired behavior in the real world. This has three consequences:

- The descriptions must be formulated in a common language both domain and software experts understand. This *communication problem* has implications for the language in which ground models are formulated, showing a possible gap between such a language and programming languages, the latter being targeted to support the execution of programs by machines, see Sect. 2.1.
- To 'explain how the solution (read: the software specification) relates to the affairs of the world it helps to handle' [55, p. 254] presents an *evidence problem* we discuss in Sect. 2.2, evidence upon which every further system verification and validation is based. The problem arises because the real-world part involved in a ground model justification has no precise contour, so that mathematical reasoning does not apply.
- The experimental character of ground model verification implies the model *validation problem* we discuss in Sect. 2.3, which is about executability of models.

Ground model software specifications represent *blueprints* of the software component which must be developed for the system in question, for good reasons called 'golden models' in the semiconductor industry. It is general engineering practice that the stakeholders, together, analyze a system blueprint, reason about its features, test its appropriateness, maybe change it repeatedly, before proceeding in common agreement to the realization of the system (and as already stated above possibly also during the development process). This is an effort to make sure that the system to-be-developed is well understood and, via its ground model, recognized as correct and complete. Therefore, as in classical engineering disciplines, also in software engineering such a blueprint provides the

[3] This is the reason why we changed the initially chosen name 'primary model' [8, p.392] into 'ground model'.

authoritative basis—a 'contract'—the software experts (designers and programmers) can rely upon for the software development via an appropriate refinement of model abstractions by code. In [70] it is nicely explained why 'there does not seem to be any compelling reason to treat the handling of models in software engineering in a radically different way than what is done in engineering in general'. This applies also to the use of ground models as blueprints, though at present using software blueprints is not common practice and contradicts the principles of agile software development. It is hard to believe that one could design blueprints for bridges, skyscrapers or airplanes in an agile way, let alone build the latter in this way: there are no early (certainly no day-to-day) deliveries of bridges, buildings or airplanes which yield continuous changes for their development. Maybe something agile sneeked into the Stuttgart main station and Berlin airport projects...

To be usable as blueprint, a ground model for the code one has to develop must have the following four properties, namely to be:

- *Precise* at the level of abstraction and rigor of the problem and of the application domain the model belongs to. This means that it contains no ambiguity which could lead the software developer to miscomprehend how the domain expert understands the model, in particular in its domain knowledge related features (which usually are not the principal domain of expertise of a software expert). See the discussion of the communication problem in Sect. 2.1.
- *Correct* in the sense that the model elements reliably and adequately convey the meaning of what in the real world they stand for. See the discussion of the evidence problem in Sect. 2.2.
- *Complete* in the sense that the model contains every behaviorally relevant system feature, but no elements which belong only to the implementation.
- *Consistent* in the sense that conflicting objectives, which may have been present in the original requirements, are resolved in the model.

It is important to be aware that the degree of detail which is required for a ground model depends on the given problem and problem domain. There is no absolut notion of precision, not even for logical systems. One cannot stress enough that what a ground model is concerned about is first of all **full understanding of what to build**, an understanding that must be shared by the stakeholders and the experts involved. In [70, Sect.2.4] such models are called *descriptive* and distinguished from *prescriptive* models, whose role is to guide the implementation (by ground model refinements, as explained in Sect. 3). For example, the ground models for an interpreter of Prolog [20], Occam [13], Java [72, Part I] and C# [15] programs are descriptive models which explain the semantics of their language at the level of programming. The corresponding virtual machine models for the WAM [21], the Transputer [12], the JVM [72, Part II-III] and the .NET CLR [34] are of prescriptive character, representing a step towards the language implementation.

Note that completeness at the ground model level means that the model contains all elements which are relevant to guarantee the intended behavior, including descriptions of what in [70, Sect.2.4] is called *primary functionality*

together with the supporting system *infrastructure* (as far as relevant for the correct realization of the primary functionality), avoiding however to introduce details which are relevant only for the implementation.

Note that for epistemological reasons, the correctness and completeness properties cannot be established by purely linguistic, system-internal means, since they relate precise conceptual features to non-verbal phenomena in the real world, as further investigated in Sect. 2.2. In contrast, when a description is precise, its consistency, a logical, system-internal property, can be checked by rigorous, scientific (for software models typically by mathematical) means.

2.1 Communication Problem

Why are ground model descriptions usually not directly programmed, but must come prior to programming?

Remember that a ground model, as blueprint of the code one has to develop, must be understandable for the two parties involved, for the software developers and for the experts of the corresponding application domain. This is a *communication problem* which is not taken seriously enough in current software engineering practice, although a thorough analysis of major accidents with software-intensive systems showed that 'The extra communication step between the engineer [read: the domain expert] and the software developer is the source of the most serious problems with software today' [51, Sect. 2.5]. The communication problem becomes only stronger where the domain experts are not familiar with mathematical notations, experts of their field but not used (neither willing to learn) to express their knowledge or reasoning in any formal logic, programming or mathematical language.

To overcome the communication problem, the language in which ground models are formulated must have the capability to calibrate the degree of precision of descriptions to the given (any) application domain, so that a direct relation can be established between the real-world items and the linguistic counterparts which represent those items in the model. This is what Leibniz called 'proportio quaedam inter *characteres* ('symbols') et *res* ('things')' and considered to be the basis of truth: 'Et haec proportio sive relatio est *fundamentum veritatis*' [50]. A ground model must allow one 'to represent the concepts of the application domain at an adequate level of abstraction such that the specialities of the applications are directly represented and not covered by awkward implementation concepts' [24, Sect. 4]. To establish such a direct association of items in the world with linguistic expressions, the language must allow the modeler to express directly, without extraneous encoding, the following:

- every kind of real-world objects with their attributes and relations, which together constitute arbitrary system 'states',
- arbitrary actions to change the set of objects or some of their properties or relations, actions which constitute arbitrary 'state changes'.

In other words, contrary to a widely held view in the Formal Methods community, the ground model language must be a clearly defined and generally

understood portion of natural language, as used in rigorous scientific and engineering disciplines, made up from precise and simple but general enough basic constructs to unambiguously represent arbitrary real-world facts (states of affairs) and state changing events. Formalized logic or programming languages are just too specific for the purpose and understood only by a rather restricted group of stakeholders, though formal languages contribute in an excellent way to do the work in appropriately circumscribed domains (see the role of domain-specific languages discussed below).

2.2 Evidence Problem for Ground Models

The fact that a ground model must establish a relation between the real-world problem and the design of the code which has to be developed brings us to the *evidence problem* for ground models, mentioned in the introduction.[4] The real-world counterpart of a ground model does not exist as an object of study, to a wide extent it resides only in the heads of the domain experts. As a consequence, there is no way to 'prove' by whatever mathematical means the appropriateness of the association of real-world objects and relations with model elements. But **model inspection** can help to provide evidence of the needed adequate correspondence between on the one side ground model elements and events, and on the other side what happens in the part of the world the model expresses. Such a review of the blueprint must be performed in cooperation between the application-domain and the software experts, whereby the two parties can check in particular the correctness and completeness of the software specification, from where the code development has to start.

Model inspection is similar to code inspection, as commonly used in current software development practice, but a) it involves not only programmers, but also domain and software design experts, b) it happens at a higher level of abstraction than that of executable code, c) it compares model elements to items and phenomena in the real world, and d) it uses domain-specific knowledge and reasoning, which only in very special cases will be formally defined. To cite an example, such model inspections were critical for the development of the ground model (an Abstract State Machine [22], ASM) for the railway process model component of FALKO, a railway timetable validation and construction system [18], which has been coded in C^{++}. The feasability of model inspection has been tested during the Dagstuhl Seminar 'Practical Methods for Code Documentation and Inspection' [16], where the ground model [17] and the source code (see [53]) for the (rather simple, industrial) Production Cell case study [52] have successfully passed an inspection session.

The epistemological role of ground model inspection, to solve the evidence problem, shows once more why the language to formulate ground models must provide means to

[4] In earlier publications we have called this a verification method problem, but since the word 'verification' carries a connotation of mathematical justification method, we move here to the more appropriate term 'evidence'.

express objects, states of affairs and events in the real world *directly*.

The appropriateness check, of the link the inspection has to investigate between the model and the world, is strongly supported by coding-free expressability, which in turn necessitates linguistic support of descriptions at whatever abstraction level. Even high-level programming languages do not suffice for that, despite of the impressive progress modern programming languages made to offer the programmers abstract concepts as commonly used in the language of mathematics, in particular logic and set theory. The reason is that each programming language is necessarily bound to a specific abstraction level, which is determined by the basic data structures, types and operations it offers.

Furthermore, modern (in particular object-oriented) programming languages, but also widely used OMG languages like UML, SysML, BPMN and other modeling languages, tend to put from the very beginning much attention on classes, their operations and structure, types, libraries, etc. which are of less importance for (and often hindering in the first phase of) system modeling (see [49] for a discussion). A similar observation has been made during the recent discussion whether coding is the new literacy, namely that 'programming as it exists now forces us to model, but it does so in an unnatural way' [39]. Also through analysing accidents of software intensive systems the need became clear that we must become 'able to grasp ... problems directly, without the intermediate muck of code' [71], to achieve an appropriate intellectual understanding of the system, given that 'Nearly all the serious accidents in which software has been involved in the past twenty years can be traced to requirements flaws, not coding errors.' [51, Sect. 2.5].

2.3 Validation Problem

The experimental character of ground model inspection brings us to the *validation problem* for ground models, which concerns models in general, including ground model refinements (see Sect. 3). A practically useful inspection procedure needs analysis support by repeatable experiments, aiming to falsify, in the Popperian sense [58], expected model behaviour, so that in case the model can be debugged before building the system. This is customary in traditional engineering disciplines. To make such an experimental validation possible for software models, these **software models should be executable**, conceptually or by machines (or be easily transformable into a machine executable version), so that runtime verification and testing become available at the application-domain level of abstraction. Model executability supports 'rapid prototyping of systems as part of the interative specification of requirements', as advocated in [23, p.15]

To support ground model inspection often the very *concept of run of a model* suffices, e.g. to check scenario behavior (confirming its correctness or revealing a conceptual mistake). A characteristic recent example for this is the process algebra model developed in [33] for the widely used Ad hoc On-Demand Distance Vector routing protocol AODV [56]. The model allowed their authors to exhibit

some conceptual misbehavior in well-known implementations of the protocol, e.g. concerning the fundamental loop freedom and route correctness.

However, if the runs can be performed mechanically, by machines and not only intellectually, that increases the debugging potential considerably. We used both techniques with advantage for analyzing the Java and JVM ground models in [72], mathematically investigating runs of the ASM models and comparing them with the corresponding AsmGofer [68] runs of the refined executable model versions and with runs of Sun's code (see [72, Ch.A]. All the bugs we discovered in this process were reported to and corrected by Sun. Another interesting example is the above mentioned development of the ground model ASM for the railway process model component of the FALKO system [18]. In this project, where the requirements came as a set of scenarios, no mathematical verification was possible, but the scenarios could be checked to validate the 'correctness' of the ground model. In fact, we validated it through AsmWorkbench [26] executions which confirmed the expectations of the scenarios. This was BEFORE we started to compile the ground model ASM to C^{++} code [69] (which has worked for years in the Vienna transportation system and never failed!).

This request for executability of software models is contrary to the still widely held view that software specifications should not be executable. The main reason which is usually given for this is that executability would imply a limitation of the expressiveness of the specification language [46]. Instead, declarative specifications are advocated,[5] using logic—axiom systems (algebraic approach) or equational theories (denotational approach)—to define the requirements, in an attempt to make specifications 'completely independent of any idea of computation' [40, p.89], 'ideally ... *predicates* on solutions' [24, Introduction]. In particular, using descriptions where abstract assignment statements $f(s_1, \ldots, s_n) := t$ occur (as is typical for ASM models [22]) were declared by highly respected colleagues to belong to implementations (see the discussion in [9]). But the question is not whether descriptions are declarative or operational, but at which level of abstraction they are formulated, as becomes clear also when one looks at B [1] or Event-B [2] or TLA+ [48] models (which formally are logical formulae but mimic the operational character of the described actions (assignments) by the x/x' notation). Different degrees of detailing serve the multiple roles of abstraction, such as providing an accurate and checkable overall system understanding, or isolating the hard parts of a system, or communicating and documenting design ideas, etc., depending on the role of every model, namely to serve a particular purpose, as emphasized in [73].

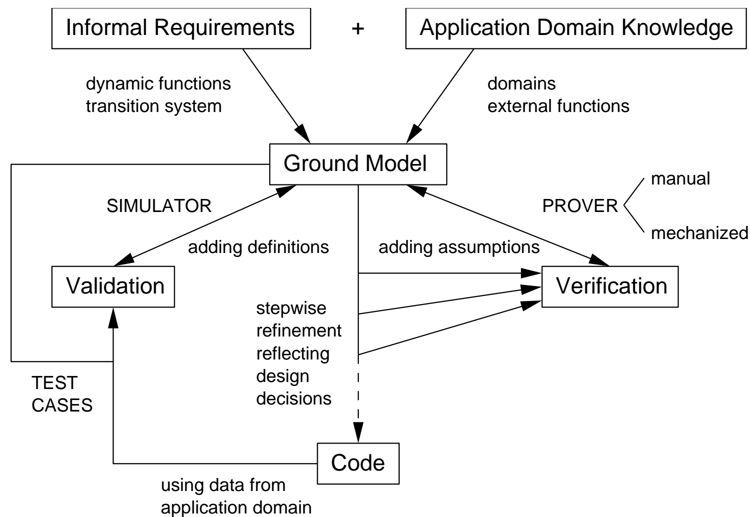
[5] In [35], which contains a detailed evaluation of the pro and contra for executable specifications, it is rightly pointed out that 'non-executable' and 'declarative' are really different terms, as illustrated by the programming interpretation of Horn clauses in logic programming languages. Note also that since Horn clauses are a reduction class of first-order logic [14, Ch.5.1], their computational interpretation is Turing complete so that using them implies no limitation of expressivity, at least in principle.

After all, model executability offers high-level debugging which allows one to detect conceptual problems low-level code inspection easily misses. For a nice (and surprising) illustrating example see [36] (or [19, 3.2]). Model executability also helps to save on the enormous cost of late code-level testing or runtime verification. Model executability also supports exploratory software development processes.

3 Refinement Method

A critical question is how to transform a ground model in a correctness preserving way to code. There are at least three different ways to do this.

- One way is to proceed right away to programming in an appropriate programming language, using the ground model as the specification of the code functionality. Justifying the correctness of the implementation is supported by code inspection which should check and document the design-intent-relation between high-level ground model and implementing code features.
  If the specification language supports modeling for change, this helps to identify and to perform the needed changes in the ground model and the corresponding code when new requirements appear. The price to pay is to keep the ground model and the code in sync, the gain is that the design-intent-relation is kept, linking ground model objects and actions to code data and code segments. This results in less maintenance cost.
  For this approach, high-level languages which offer explicit expressions or instructions for abstract modeling concepts simplify the programming and justification task enormously. A strong support, as advocated by Language Oriented Programming [74,28] and supported by the Meta Programming System [54], comes also from domain-specific languages (DSLs) whose constructs directly reflect specific application domain concepts. This holds also if the DSL is directly implemented in a high-level programming language, see some characteristic examples in [44,3,45]. Methodologically speaking, plugins in CoreASM [25], which have to be programmed in Java, provide an analogous support for domain-specific extensions of executable ASM models.
- A second way is to write a compiler for a class of ground models that includes those one has to expect in the given application domain. For two examples see [5,18]. Justifying the correctness of model implementations consists in showing that the compiler correctly translates ground model items (objects, events, actions) to code. This compiler correctness needs to be shown only once, by verification experts, so that the correctness problem for the code which implements a concrete ground model is reduced to the ground model correctness problem—a fruitful application of the divide-and-conquer principle.
- A third way is to use stepwise model refinement to turn the descriptive ground model into a prescriptive implementation model (in the sense defined in [70]), leading to code. This approach helps to reduce the gap between the ground model and the level of detail needed for the target code. To



guarantee preservation of the design intent, each refinement step has to be checked to be correct; the verification should be documented to be controllable. Executability of refined models supports their experimental validation by simulations.
The refinements (and their justification) are the job of the software engineer, including the special case where a refinement step reveals the necessity to correct or complete the ground model in cooperation with the domain experts. The application-domain expert has to rely upon the professionality of the implementor who understands all the details of the language used for model refinements and code. The practicality of this approach depends on the refinement concept. The figure above[6] illustrates the structure of such a software development process.
It is a common experience (and pointed out also in [70]) that inconsistencies and conflicts between different design proposals often are only detected in fine-grained models. Their resolution has to backtrack in the refinement chain, possibly coming back to the ground model. For this reason, the completeness and correctness of the ground model are guaranteed only once the final code is ready; the *process* of building the ground model and its refinements is by no means linear. But to support the maintenance and evolution of software efficiently, it is important that all models involved by the code are fully documented, each one reflecting corresponding design decisions, and are kept in sync.

[6] © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission, copied from [22].

4 Conclusion: Refinement Patterns

This is not the place to evaluate which modeling methods support building satisfactory ground models. But we want to formulate a challenging problem, hoping that it may attract somebody. It concerns mainly, but not only, the Abstract State Machines method [22] which has been used with success to build ground models, and their refinements, for various real-life systems, industrial standards and semantics of programming languages (for surveys see [22, Ch.9], [19, p.4] and [11], respectively).

A characteristic feature of the ASM method is its refinement notion [10], which has been developed originally for a modular definition of the semantics of Prolog (see [6,7,8]). ASM refinements allow one to refine not only data, but also operations (rules) and more generally entire segments of computation. This feature offers the designer ways to describe design ideas directly in terms of ASM refinement steps, but it also makes the notion considerably more general and harder to implement than the various refinement concepts in other state-based specification methods, e.g. B [1], Event-B [2], TLA$^+$ [48] and Z [27] (among others). These methods come with carefully restricted refinement definitions so that the refinements are supported by the corresponding proof tools.

There is an unavoidable trade-off between generality of the refinement notion and the degree to which it can be supported by theorem provers. Gerhard Schellhorn has thoroughly investigated the ASM refinement notion in [59,60,62,61,63] and has implemented it in the KIV [47] theorem prover. The original goal was to machine check, using KIV [64,65], the WAM correctness proof in [21]; this proof proceeds via a hierarchy of a dozen proven-to-be-correct ASM refinement steps, starting with a ground model ASM [20] for the ISO standard of Prolog and ending with a complete model of the WAM. In the sequel, KIV verification has been applied to various other ASM models using the ASM refinement mechanism. Two major examples are the electronic purse case study Mondex [42] and the flash file system verification [66]. The Mondex example illustrates the practical feasibility of the ASM refinement approach for code verification: the refinement from the transaction to the protocol level [67] deals with the original Mondex case study, but the authors could also add a refinement to the security level (with crypto primitives) [43,41] and a refinement from there to Java code [38]. It is interesting to note that for the verification of the flash file system the refinement theory had to be developed further to handle not only functional correctness, but also crash safety, down to the level of code (see [30,57]). In particular [31], which further refines the high-level models in [32], illustrates the crucial role of finding appropriate intermediate models to verify that low-level features, like recovery from unexpected power cuts, are guaranteed.

The ASM refinement notion has been implemented also in PVS [29] where it has been used for compiler correctness proofs, starting with ASM ground models for source and target languages [37,75].

Here is the challenge: is it possible to *distill practically useful refinement patterns which come with corresponding compositional and tool supported proof patterns*, using the general ASM refinement concept? I addressed this question

to colleagues in the theorem proving community when in [4] we realized that the Java2JVM compilation correctness theorem, as stated and proved for stepwise refined ASM models of Java/JVM in [72], could have been proved by instruction-wise refinement steps where each new instruction is accompanied by a modular proof extension. This supports modeling for change and software product lines in a rather strong way. Can such a modular (in logical terms 'conservative') proof extension which corresponds to an ASM refinement step be supported by current theorem provers and to what extent?

Acknowledgement. We thank the following colleagues for a critical reading of various drafts of the paper: Heinz Dobler, Alexander Raschke, Klaus-Dieter Schewe.

To appear in: M. Broy, K. Havelund, R. Kumar, B. Steffen (eds): Towards a Unified View of Modeling and Programming (Proc. Isola 18), Springer LNCS

References

1. J.-R. Abrial. *The B-Book*. Cambridge University Press, Cambridge, 1996.
2. J.-R. Abrial. *Modeling in Event-B*. Cambridge University Press, 2010.
3. C. Artho, K. Havelund, R. Kumar, and Y. Yamagata. Domain-specific languages with scala. In S. C. M. Butle and and F. Zaidi, editors, *Proc. 17th International Conference on Formal Engineering Methods*, volume 9407 of *LNCS*, pages 1–16. Springer, 2015.
4. D. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. Universal Computer Science*, 14(12):2059–2082, 2008.
5. G. Berry. Formally unifying modeling and design for embedded systems - a personal view. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications. ISoLA 2016*, volume 9953 of *Lecture Notes in Computer Science*, pages 134–149, Cham, 2016. Springer.
6. 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.
7. 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.
8. 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.
9. E. Börger. Review of E. W. Dijkstra and C. S. Scholten *Predicate Calculus and Program Semantics* (Springer-Verlag 1989). *Science of Computer Programming*, 23:1–11, 1994.
10. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
11. E. Börger. The Abstract State Machines method for modular design and analysis of programming languages. *J. Logic and Computation*, 2014.

12. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
13. 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.
14. E. Börger, E.Grädel, and Y.Gurevich. *The Classical Decision Problem.* Perspectives in Mathematical Logic. Springer-Verlag, 1997. Second printing in "Universitext", Springer-Verlag 2001.
15. 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.
16. E. Börger, P. Joannou, and D. L. Parnas. *Practical Methods for Code Documentation and Inspection*, volume 178. Dagstuhl Seminar No. 9720, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, May 1997.
17. E. Börger and L. Mearelli. Integrating ASMs into the software development life cycle. *J. Universal Computer Science*, 3(5):603–665, 1997.
18. 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.
19. E. Börger and A. Raschke. *Modeling Companion for Software Practitioners*. Springer, 2018. For Corrigenda and lecture material on themes treated in the book see http://modelingbook.informatik.uni-ulm.de.
20. E. Börger and D. Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24:249–286, 1995.
21. 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.
22. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
23. F. P. Brooks. No silver bullet: Essence and accidents of software engineering,. *Computer*, 20(4):10–19, 1987.
24. M. Broy, K. Havelund, and R. Kumar. Towards a unified view of modeling and programming. In T. Margaria and B. Steffen, editors, *7th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation-Part II. (ISoLA 2016)*, volume 9952 of *Lecture Notes in Computer Science*, pages 238–260, Cham, 2016. Springer.
25. The CoreASM Project. http://www.coreasm.org and https://github.com/coreasm/, since 2005.
26. 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. Published in HNI-Verlagsschriftenreihe, Vol. 83.
27. J. Derrick and E. A. Boiten. *Refinement in Z and Object-Z*. Springer, 2001.
28. S. Dmitriev. Language oriented programming: The next programming paradigm. *onBoard Electronic Monthly Magazine*, April 2010. http://www.onboard.jetbrains.com/articles/04/10/lop/index.html.
29. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.

30. G. Ernst and J. Pf¨ Modular, crash-safe refinement for asms with submachines. *Science of Computer Programming*, 131:3 – 21, 2016. Abstract State Machines, Alloy, B, TLA, VDM and Z (ABZ 2014).
31. G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif. Inside a verified flash file system: Transactions and garbage collection. In A. Gurfinkel and S. A. Seshia, editors, *Verified Software: Theories, Tools, and Experiments*, pages 73–93, Cham, 2016. Springer International Publishing.
32. G. Ernst, G. Schellhorn, D. Haneberg, J. Pfähler, and W. Reif. Verification of a Virtual Filesystem Switch. In *Proc. of Verified Software: Theories, Tools, Experiments (VSTTE)*, volume 8164 of *LNCS*, pages 242–261. Springer, 2013.
33. A. Fehnker, R. van Glabbeek, P. Hoefner, A. McIver, M. Portmann, and W. L. Tan. A process algebra for wireless mesh networks used for modelling, verifying and analysing AODV. Technical Report 5513, NICTA, Brisbane (Australia), 2013.
34. N. G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2006.
35. N. E. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, 1992.
36. V. Gervasi and E. Riccobene. From English to ASM: On the process of deriving a formal specification from a natural language one. In *Integration of Tools for Rigorous Software Construction and Analysis*, volume 3(9) of *Dagstuhl Report*, pages 85–90, 2014. Dagstuhl Seminar 13372 organized by Uwe Glässer, Stefan Hallerstede, Michael Leuschel, Elvinia Riccobene, 08.–13.9.2013. DOI: 10.4230/DagRep.3.9.74, URN: urn:nbn:de:0030-drops-43584, URL: http://drops.dagstuhl.de/opus/volltexte/2014/4358/.
37. 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.
38. 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.
39. C. Granger. Coding is not the new literacy. http://www.chris-granger.com/2015/01/26/coding-is-not-the-new-literacy/, January 2015. Consulted 01/12/2017.
40. J. A. Hall. Taking Z seriously. In *ZUM'97*, volume 1212 of *Lecture Notes in Computer Science*, pages 89–91. Springer-Verlag, 1997.
41. D. Haneberg, H. Grandy, W. Reif, and G. Schellhorn. Verifying smart card applications: An ASM approach. In *Proc. Conference on Integrated Formal Methods (iFM 2007)*, volume 4591 of *LNCS*. Springer, 2007.
42. 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.
43. D. Haneberg, G. Schellhorn, H. Grandy, and W. Reif. Verification of Mondex electronic purses with KIV: from transactions to a security protocol. *Formal Aspects of Computing*, 20(1):41–59, 2008.
44. K. Havelund. Data automata in scala. In M. Leucker and J. Wang, editors, *Proc. 8th International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 1–9. IEEE Computer Society Press, 2014.
45. K. Havelund and R. Joshi. Modeling and monitoring of hierarchical state machines in scala. In *Proc. 9th International Workshop on Software Engineering for Resilient Systems (SERENE 2017)*, Springer LNCS, Geneva (CH), September 2017.

46. I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, 4(6):330–33, November 1989.
47. The KIV System. http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/kiv/.
48. L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. Available at http://lamport.org.
49. L. Lamport and L. C. Paulson. Should your specification language be typed? *ACM Transactions on Programming Languages and Systems*, 21(3):502–526, May 1999.
50. G. W. Leibniz. Dialogus de connexione inter res et verba. G. W. Leibniz: Philosophische Schriften, August 1677. Edited by Leibniz-Forschungsstelle der Universität Münster, Vol.4 A, n.8. Akademie Verlag 1999.
51. N. G. Leveson. *Engineering a Safer World: Systems Thinking Applied to Safety*. Engineering Systems. MIT Press, 2012.
52. C. Lewerentz and T. Lindner. *Formal Development of Reactive Systems. Case Study "Production Cell"*, volume 891 of *Lecture Notes in Computer Science*. Springer, 1995.
53. L. Mearelli. Refining an ASM specification of the production cell to C++ code. *J. Universal Computer Science*, 3(5):666–688, 1997.
54. Meta Programming System. https://www.jetbrains.com/mps/.
55. P. Naur. Programming as theory building. *Microprocessing and Microprogramming*, 15, 1985.
56. C. Perkins, E. Belding-Royer, and S. Das. Ad hoc on-demand distance vector (AODV) routing. Technical Report RFC 3561, Copyright (C) The Internet Society, Network Working Group, July 2003. http://tools.ietf.org/html/rfc3561.
57. J. Pfähler, G. Ernst, S. Bodenmüller, G. Schellhorn, and W. Reif. Modular verification of order-preserving write-back caches. In N. Polikarpova and S. Schneider, editors, *Integrated Formal Methods*, pages 375–390, Cham, 2017. Springer International Publishing.
58. K. Popper. *Logik der Forschung*. Springer, 1935.
59. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
60. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
61. G. Schellhorn. ASM refinement preserving invariants. *J. UCS*, 14(12):1929–1948, 2008.
62. G. Schellhorn. Completeness of ASM refinement. *Electr. Notes Theor. Comput. Sci.*, 214:25–49, 2008.
63. G. Schellhorn. Completeness of fair ASM refinement. *Sci. Comput. Program.*, 76(9):756–773, 2011.
64. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
65. 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.
66. G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif. Development of a verified flash file system. In *Proc. of Alloy, ASM, B, TLA, VDM, and Z (ABZ)*, volume 8477 of *LNCS*, pages 9–24. Springer, 2014.
67. 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.
68. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at https://tydo.eu/AsmGofer.
69. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.
70. B. Selic. Programming $\subset$ Modeling $\subset$ Engineering. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, volume 9952 of *Lecture Notes in Computer Science*, pages 11–26, Cham, 2016. Springer.
71. J. Somers. The coming software apocalypse. The Atlantic, September 26 2017. Email newsletter, consulted on November 11, 2017.
72. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
73. B. Thalheim and I. Nissen, editors. *Wissenschaft und Kunst der Modellierung: Kieler Zugang zur Definition, Nutzung und Zukunft*, volume 64 of *Philosophische Analyse / Philosophical Analysis*. De Gruyter, 2015.
74. M. P. Ward. Language oriented programing. *Software - Concepts and Tools*, 15(4):147–161, 1994.
75. 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.