The ASM Ground Model Method as a Foundation of Requirements Engineering

Egon Börger

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

Abstract. Building ground models is one of the three constituents of the engineering method for computer-based systems which is known as Abstract State Machine (ASM) method [16]. In this note we characterize *ground models*, whose epistemological role for a foundation of system design resembles the one Aristotle assigned to axioms to ground science in reality, avoiding infinite regress. We explain how ASM ground models help to resolve two major problems of requirements engineering, providing means a) to obtain for complex computer-based systems an *adequate understanding* by humans, and b) to *cope with ever-changing requirements* by faithfully capturing and tracing them via well-documented modeling–for–change. We point out that via an appropriate refinement method one can relate ground models to executable code.

1 IINTRODUCTION

In a recent paper [6] Daniel Berry identifies requirements engineering as the main source for "the inevitable pain of software development", explaining "why there is no silver bullet" [17]. We agree that the development of good application-software design for complex computer-based systems is a hard, intellectually demanding activity which is as little mechanizable as for example proving interesting mathematical theorems[1]. We also agree that the main difficulty is encountered when it comes to define (read: specify and design) what Brooks called "the conceptual construct" or the "essence" of software, as opposed for example to its representation by code. Defining what the software for a computer-based system is supposed to do takes place mainly during the requirements engineering phase[2] where two stumbling blocks are to be overcome, namely

- to help ensure a correct understanding by humans,
- to cope with the continuous change of the requirements.

[1] This does not preclude the automation of key tasks of application design, once such tasks have been identified as recurring patterns. The situation can be compared to the automization of proof principles in computer-assisted proof systems.

[2] We agree with Dave Parnas [14] that the word "engineering" in this context is bombastic; " requirements capture, analysis and documentation" would probably be more adequate.

We argue in this note that building *ground models* as introduced in [7–9][3], describing requirements by *Abstract State Machine* (ASM[4]) models, provides a practical methodical support to tackle those two problems in a way which scales to large industrial systems. Furthermore, combining the construction of ASM ground models with the stepwise refinement method of ASMs described in [12] allows one to link requirements models to executable code in an organic and effectively maintainable chain of rigorous and coherent system models which captures and documents design decisions made at successive levels of detailing.

This note is of purely methodological nature so that for the involved technical definitions, results and illustrating examples we provide references to the literature. In Section 2 we explain that ground models are inevitably present in every system design, the question being whether or not this happens in an explicit form which allows humans to understand and analyze the ground model, in particular to satisfactorily relate it to the given requirements. We characterize ground models by their basic properties and compare their role for a trustworthy foundation of system software to the role Aristotle assigned to axioms as a foundation for scientific theories. In Section 3 we explain why among numerous other concepts ASMs are particularly well suited as a scientific foundation for satisfactory ground models. In Section 4 we point out that applying to ASM ground models the ASM refinement method, which generalizes Wirth's and Dijkstra's classical refinement approach [32, 19] by a most general notion of refinement described in [12], offers the designer a practical support to capture, document and trace *changing* requirements from the abstract application-domain level down to executable code.

The combination of ASM ground model construction and of sucessive model refinements, which is known as the ASM method and is presented in [16], is not a "silver bullet", in the sense that it does not provide any mechanizable algorithmic way of building good systems. But due to the basic character of the concepts involved, the ASM method provides an accurate support for what the practitioner in one way or the other has to do anyway to achieve a sufficient understanding and a reliable management of changing requirements. Since in addition those basic concepts appear in the ASM method in a technically simple form, they can be learned without any special training (see the appendix where the basic definitions are listed or consult [16] for a textbook style introduction). Therefore the ASM method can be adopted as a professional discipline which helps in identifying and analyzing "the conceptual construct" of computer-based systems and turning creative ideas in a traceable way into the architecture of truly trustworthy, evolving software systems.

2 THE ROLE OF GROUND MODELS

The fundamental problem is clearly identified in Brooks' famous analysis in [17]:

[3] In those papers ASMs were used to build a ground model for what later became the ISO standard of Prolog (see [13]), but the used method is independent of Prolog.

[4] See the appendix for a definition.

> The hardest single part of building a software system is deciding precisely what to build.

It is the role of requirements to describe "what to build", but too often their formulation is incomplete or too detailed, ambiguous or inconsistent. When the requirements fail to provide an accurate understanding of the system to be built, a typical escape is to consider the code as the true definition of the system. However, this does not solve the problem of "grounding the design in reality". In fact in this way no correspondence is provided between the extra-logical theoretical terms appearing in a program and their empirical interpretation, violating a basic principle of Carnap's analysis of scientific theories in [18]. In addition taking the code as system definition makes it hard to faithfully reflect changing high-level requirements and to document them in a transparent way.

The notoriously difficult and error prone elicitation of requirements is largely a *formalization* task in the sense of an accurate task formulation, namely to realize the transition from usually natural-language problem descriptions to a sufficiently precise, unambiguous, consistent, complete and minimal formulation of what Brooks [17] calls "the conceptual construct" or the "essence" of a computer-based system, as distinguished from a software representation, e.g. by code. We use the term *ground model*[5] for such formulations of "the conceptual construct", "blueprints" of the to-be-implemented piece of "real world" which "ground the design in the reality". By its very epistemological role of relating a non-linguistic reality to a linguistic description, the concept of ground model has no purely mathematical definition, though it can be given a scientific definition in terms of basic epistemological concepts which have been elaborated for empirical sciences by analytic philosophers[6]. To illustrate the point we want to make on ASM ground models, it suffices to list the essential properties every satisfactory scientific substitute for the intuitive concept has to possess. Before doing this, we shortly characterize three basic problems ground models have to solve.

2.1 Three problems for any formalization effort

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, where the relevant piece of reality has to be represented. This is mainly a *language and communication problem* between the domain expert or customer and the software designer who prior to coding have to come to a common understanding of "what to build", to be documented in a contract containing a model which can be inspected by the involved parties. 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 and to be easily understandable by the two parties involved. This includes the capability to calibrate the degree of precision of the language to the

[5] *Ground models* [10] were originally called *primary models* [9, Sect. 3].
[6] For one such proposal we refer to [22, 23].

given problem, so as to support the concentration on domain issues instead of issues of notation. It also means that the modeling 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).

The second formalization problem 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 passage from an informal to a precise description. Every 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, 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 reality the model is supposed to capture. This is analogous 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[7].

Two kinds of means are needed to establish that a ground model is complete and consistent, that it reflects the original intentions and that these are correctly conveyed – together with all the necessary underlying application-domain knowledge – to the designer. To check the completeness property, which is clarified further below, it must be possible to proceed by inspection of ground models by the application-domain expert[8]. But also appropriate forms of domain-specific reasoning, not limited to formal deductions in *a priori* determined logic systems, have to be available to support the designer in formally checking the internal consistency of the model, as well as the consistency of different system views. Such a 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, e.g., [31, 24]).

The third formalization problem is a *validation problem*. It must be possible to perform experiments with the ground model, in particular to simulate it for running relevant scenarios (use cases), providing a framework for system-

[7] Certainly the epistemological status of the underlying concept of evidence has to 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 [3].

[8] Providing a precise ground against which questions can be formulated, ground models support the Socratic method of asking "ignorant questions" [4] to check whether the semantic interpretation of the informal problem description is correctly captured by the mapping to the terms in the mathematical model.

atic attempts to "falsify" the model in the Popperian sense [28] against the to-be-encoded piece of reality. This empirical criterion also takes into account that computer-based systems are not purely intellectual artefacts but inserted in a real-world environment. 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 [25]), this is captured by the corresponding run segments (simulations) in the ground model. 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.

2.2 Intrinsic Properties of Ground Models

To be appropriate as high-level models for complex real-life systems, also under industrial constraints, ground models have to be

- *precise* at the appropriate level of detailing yet *flexible*, to satisfy the required accuracy exactly, without adding unnecessary precision;
- *simple and concise* to be understandable by both domain experts and system designers and to be manageable for inspection and analysis, avoiding any extraneous encoding and through their abstractions "directly" reflecting the structure of the real-world problem. These two properties help to uncover requirements one may initially not be aware of;
- *abstract (minimal) yet complete*. Completeness means that every semantically relevant feature is present, that all contract benefits and obligations are mentioned and that there are no hidden clauses. In particular, a ground model must contain as interface all semantically relevant parameters concerning the interaction with the environment, and where appropriate also the basic architectural system structure. The completeness property "forces" the requirements engineer, as much as this is possible, to produce a model which is "closed" modulo some "holes", which are however explicitly delineated, including a statement of the assumptions made for them at the abstract level and to be realized through the detailed specification left for later refinements. Model closure implies that no gap in the understanding of "what to build" is left, that every relevant bit of implicit domain knowledge has been made explicit. The completeness property should not preclude modular ground model descriptions. Minimality means that the model abstracts from details that are relevant either only for the further design or only for a portion of the application domain which does not influence the system to be built;
- *validatable* and thus in principle falsifiable by experiment, satisfying the basic Popperian criterion for scientific models [28];

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

3 HOW TO SOLVE THE FORMALIZATION PROBLEM

Using Traditional "Formal Methods". With respect to the communication and verification problem most so-called formal methods miserably fail as satisfactory candidates for building *ground* models, because they propose formalizations in the necessarily restrictive syntax of some logic, or verifications by a-priori-fixed rule-based (in many cases mechanizable) reasoning schemes, which go beyond the expertise that can reasonably be expected from software practitioners or domain experts without a thorough training in mathematical logic. Many formal methods also fail with respect to the validation problem, due to their intendedly declarative non-executable character which is intrinsic for axiomatic specifications. We doubt whether "Formal Methods are best applied during requirements engineering" [5]. Most of the successful formal methods, 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 [2, 1]. Another good reason for using particular "formal methods" may be to encapsulate some specific design experience, turning so to speak part of the design into a language-supported (possibly mechanic) procedure. To cite an example we can refer to the SCR method [26], or the Requirements State Machine model defined in [27], which relates process-control systems to methods for checking a set of criteria identifying missing (as well as incorrect or ambiguous) requirements.

Using Abstract State Machines. The proposal to use Abstract State Machines as precise mathematical form of ground models goes back to [7–9] where it was used to define what later became the ISO standard of Prolog. In fact, ASMs are not "formal" in the just mentioned restricted understanding of the term, but instead support intuitive, content-oriented, precise modeling and reasoning as used in mathematical and experimental sciences. A basic ASM as defined in [20] comes as set of "guarded commands", more precisely rules of form **if** *Condition* **then** *Updates*, which at each step are fired simultaneously to update the current abstract state. Besides this general form of rules, the language of ASMs is unconstrained, as is the set of reasoning principles. The definition and its natural extension to asynchronous ASMs—which obviously can be made more rigorous and even formalized, see the appendix and the Asm-Book [16, Ch.2]—accurately supports the way domain experts use high-level process-oriented descriptions and software practitioners use pseudo-code.

ASMs solve the language and communication problem due to the broad-spectrum, basic algorithmic character of the language of ASMs, which is easy to understand for domain experts, not only for programmers, and allows one to tailor the ground model to resemble the structure of the real-world problem. This works for algorithmic processes and general procedures in whatever domain, whether to compute and implement mathematical functions or database and network management processes or the control of microprocessors and technical devices or business procedures, etc. (see, e.g., the variety of examples in Chapters 3,5,6 of [16]). It includes continuous dynamic environmental features of hybrid systems, which can be captured in various ways in this ASM framework, for example by exploiting the possibility to define parts of the system in a purely mathematical (functional or declarative) way, e.g. by differential equations. For an alternative way to model real-time system behavior with infinitesimal exactness and refinements of actions by interleaved ASMs see [29]. Similarly, interaction features can be included by an appropriate use of external functions computed by different agents (see the ASM function classification in [11] or the appendix), or by adapting the standard notion of sequential ASM run to partial-order runs of asynchronous ASMs (see [21] and [16, Ch.6]).

Using ASMs as ground models also solves the verification problem since it allows one to use both inspection – for checking the model correctness and completeness with respect to the problem to be solved – and reasoning to analyze its consistency, using whatever reasoning means are appropriate. The notation does not limit the verification space. 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.

The validation problem is solved by the operational character of ground model ASMs, which come with a standard notion of computation or "run". Simulations of ground models are possible by mental simulation or using various tools which make large classes of ASMs executable (see [16, Chapter 8]). In addition, the operational character of ASMs supports defining in abstract run-time terms the expected system effect on samples – the so-called *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. Furthermore, ASM ground models can be used 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 test cases from use cases. Last but not least, by appropriately refining the oracle, one can also *specify and implement a comparator* by determining for runs of the ground model and the 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 point to in the next section.

ASMs can be made to possess all the properties listed above for ground models. Ground models built using other formal methods do share some of these properties, but usually not all of them (see [10] for a detailed comparison). The complementary properties of being *precise* and *flexible* can be obtained by fine-tuning the ASMs to the needed degree of rigor, with the consequence that the resulting ASM ground models are adaptable to different application domains and easily modifiable or extendable for reuse for changing requirements. The completeness property of ASM ground models does not preclude modular ground model descriptions, using composition and submachines concepts as defined in [15]. The *operational* character of ASM ground models supports their process-oriented understanding, their mental or machine simulation, and the possibility to turn ground models into prototypes.

The above listed properties of ground model ASMs make them also fit for reuse in the sense that when building a system, the "right" abstractions can be expressed directly, avoiding any extraneous coding, to tailor the possibly changing system specification to the characteristic conceptual frame of the underlying application-domain problem and to the desired level of detail. In this way, during the attempt to understand and faithfully capture both the system to be built and the relevant application domain assumptions, using ASM ground models supports "asking the right questions" [5, Sect.7] and annotating the answers without any hindering formal-language-constraint (see in [16] Section 3.1.1 on "Fundamental questions to be asked" when building ASM ground models)[9].

In particular, when the requirements change, most often by incrementally incorporating new features which are consistent with the given ones, these changes can be directly reflected by adaptations of the ground model abstractions, in the incremental case by conservative extensions. Tracing changes of requirements in the code is related to tracing the appropriate changes in model refinements, a complementary method we refer to in the next section. Thus each one of a set of ground models built during an entire system life cycle represents one stage of the requirements, so that "freezing" a set of requirements in one model does not prevent changing that set and formalizing it by a refined ground model. Another source for a possible multiplicity of different ground models stems from the usually many ways to describe the desired system by abstracting from specific details. This is like saying that there are many equivalent ways to formulate a contract.

Nothing in the ASM method can *guarantee* an "appropriate" application of this abstraction and refinement mechanism, which is inherent in the concept of ASMs, though by their unconstrained nature ASMs are flexible enough to be put to use as part of any methodological requirements capture and system

[9] In this context the analogy to mathematics is illuminating. Asking the right questions is by far the most important part of mathematical research, certainly prior to the invention of new methods which solve important problems. The "openness" of the language of mathematics, of which the equally open algorithmic language of ASMs is a part, is crucial for the possibility to formulate such questions against a not fully formalized yet sufficiently precise, application-domain-determined background.

design scheme. We have no "silver bullet" to offer, no fool-proof button-pushing or mechanical design and verification procedure, but the ASM method directly supports professional knowledge and skill in "building models for change". This is the best one can hope for, given the intrinsically creative character of defining "the conceptual construct" of a complex computer-based system.

4 REFINEMENT: MAKING CHANGING REQUIREMENTS TRACEABLE

The adaptation of ASM ground models to changing requirements is supported by a generalization of Wirth's and Dijkstra's classical refinement method [32, 19], based upon the use of ASMs, introduced for the first time in [7–9]. This generalization of the refinement method for ASMs, which in fact is a companion to the ASM ground model method, has been described in [12] so that in this note we mention only the fact that it is characterized by the *practical* forms of refinement it supports. They provide means to reflect successive design decisions in a traceable way via stepwise refined models which add more and more details and link ground models to executable code. 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 the syntax-directed refinement notions of the B-method [2]. ASM refinements provide a documentation of the entire design which supports design reuse and code maintenance. They do this in a simple way the practitioner can use in his daily work, for reasons which were 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.

It is an often heard objection that in practice it will be difficult to effectively maintain a chain of stepwise refined models with more than 2 levels, given that related documents "eventually and unavoidably" get out of sync. Or even more it is claimed, as was mentioned already above, that the code should be considered as the true definition of the system. Using a chain of stepwise refined models changes this widely deplored state-of-the-art situation and has the potential to enhance the designer's activity, enabling him to exactly localize the "right" level of abstraction where the desired change has to be performed and from where it has to be transfered to the more detailed lower levels. A good refinement strategy aims at encapsulating orthogonal features. 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 any way, but to do it step by step and not in one blow. This makes it easier to understand the changed implementation details and to control their effect on the entire system. If there are only two such levels, it means that the executable code is

rather close to the requirements level, but this usually happens only for simple systems. The AsmBook [16] contains numerous examples and case studies for refinement hierarchies of more than two levels. For the most advanced real-life public-domain case study which involves all the ground model and refinement capabilities offered by the ASM method, for both design and analysis, see the Jbook [30] which is focussed on modeling and verifying the semantics of Java and its implementation by the Java Virtual Machine.

Acknowledgement

This paper has been triggered by a discussion with Daniel Berry on the ASM method, in particular by the two recent papers [5] and [6]. We thank also the following colleagues for discussing and critically commenting upon draft versions of this paper: D. Batory, D. Bjoerner, J. Bowen, U. Glässer, H. Rust, M. Sintzoff, G. Snelting, T. Maibaum.

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. *The B-Book*. Cambridge University Press, 1996.
3. D. Barnocchi. L''Evidenza' nell'assiomatica aristotelica (contributo all'interpretazione dell'assiomatica aristotelica alla luce della moderna logica matematica). *Proteus*, 5:133–144, 1971.
4. D. M. Berry. The importance of ignorance in requirements engineering. *J. of Systems and Software*, 28(2):179–184, 1995.
5. D. M. Berry. Formal methods: the very idea. some thoughts on why they work when they work. *Science of Computer Programming*, 42(1):11–27, 2002.
6. D. M. Berry. The inevitable pain of software development: Why there is no silver bullet. In *Proceedings of Monterey Workshop 2002: Radical Innovations of Software and Systems Engineering in the Future*, October 2002.
7. 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 *LNCS*, pages 36–64. Springer-Verlag, 1990.
8. 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.
9. 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, the Netherlands, 1994.
10. E. Börger. Why use evolving algebras for hardware and software engineering? In M. Bartosek, J. Staudek, and J. Wiederman, editors, *Proceedings of SOFSEM'95, 22nd Seminar on Current Trends in Theory and Practice of Informatics*, volume 1012 of *LNCS*, pages 236–271. Springer-Verlag, 1995.

11. 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)*, number 1641 in LNCS, pages 1–43. Springer-Verlag, 1999.
12. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 14, 2003, to appear.
13. 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.
14. E. Börger, B. Hörger, D. L. Parnas, and D. Rombach. *Requirements Capture, Documentation, and Validation*, volume 241. Dagstuhl Seminar No. 99241, June 1999.
15. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *LNCS*, pages 41–60. Springer-Verlag, 2000.
16. E. Börger and R. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
17. F. P. J. Brooks. No silver bullet. *Computer*, 20(4):10–19, 1987.
18. 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.
19. E. Dijkstra. Notes on structured programming. In *Structured Programming*, pages 1–82. Academic Press, 1972.
20. Y. Gurevich. Evolving algebras. A tutorial introduction. *Bulletin of EATCS*, 43:264–284, 1991.
21. Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
22. A. M. Haeberer and T. S. E. Maibaum. Scientific rigour, an answer to a pragmatic question: a linguistic framework for software engineering. Number 23 in International Conference on Software Engineering, Toronto, 2001.
23. A. M. Haeberer, T. S. E. Maibaum, and M. V. Cengarle. Knowing what requirements specifications specify. Typoscript, 2001.
24. A. Hall. Seven myths of formal methods. *IEEE Software*, pages 11–19, September 1990.
25. 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.
26. C. Heitmeyer. Using SCR methods to capture, document, and verify computer system requirements. In E. Börger, B. Hörger, D. L. Parnas, and D. Rombach, editors, *Requirements Capture, Documentation, and Validation*. Dagstuhl Seminar No. 99241, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 1999.
27. 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.
28. K. Popper. *Logik der Forschung. Zur Erkenntnishtoeire der modernen Naturwissenschaft*. Wien, 1935.
29. H. Rust. *A non-standard approach to operational semantics for timed systems*. Habilitation thesis, BTU Cottbus, Germany, 2003.
30. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001. .

31. J. M. Wing. A specifier's introduction to formal methods. *Computer*, pages 8–24, Sept. 1990.
32. N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 1971.

5 APPENDIX: ASMS IN A NUTSHELL

In this appendix we list the basic definitions concerning ASMs as far as needed to work with them, covering the definition from [21] and the classification of functions from [10, 11]. For a more detailed definition of these terms and an illustration by characteristic examples we refer the interested reader to Section 2.4 of the AsmBook [16].

5.1 Basic ASMs and their Runs

An ASM is a finite set of so called *transition rules* of form

if *Condition* **then** *Updates*

which transform abstract states. Two more forms are explained below. The *Condition* (also called *guard*) under which a rule is applied is an arbitrary predicate logic formula without free variables, whose interpretation evaluates to *true* or *false*. *Updates* is a finite set of assignments of form $f(t_1, \ldots, t_n) := t$ whose execution is to be understood as *changing* (or defining, if there was none) in parallel the value of the occurring functions f at the indicated arguments to the indicated value. More precisely, in the given state first all parameters t_i, t are evaluated to their values, say v_i, v, then the value of $f(v_1, \ldots, v_n)$ is updated to v which represents the value of $f(v_1, \ldots, v_n)$ in the next state. Such pairs of a function name f, which is fixed by the signature, and an optional argument $(v_1, \ldots, v_n)$, which is formed by a list of dynamic parameter values v_i of whatever type, are called *locations*. Location-value pairs (loc, v) are called *updates*.

The notion of ASM *states* is the classical notion of mathematical *structures* where data come as abstract objects, i.e., as elements of sets (also called *domains* or *universes*, one for each category of data) which are equipped with basic operations (partial *functions* in the mathematical sense) and predicates (attributes or relations). For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used. Without loss of generality we usually treat predicates as characteristic functions and constants as 0-ary functions. Partial functions are turned into total functions by interpreting $f(x) = undef$ with a fixed special value *undef* as $f(x)$ being undefined.

The notion of ASM *run* is an instance of the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing *simultaneously* all updates of all transition rules whose guard is true in the state, if these updates are consistent, in which case the result of their execution yields the *next* state. In the case of inconsistency the computation does not yield a next state, a situation which typically is reported by executing

engines with an error message. A set of updates is called *consistent* if it contains no pair of updates with the same location, i.e. no two elements $(loc, v), (loc, v')$ with $v \neq v'$. An ASM step is performed as an atomic action with no side effects.

Simultaneous execution provides means to locally describe a global state change, namely as obtained in one step through executing a set of updates. The only limitation—imposed by the need of uniquely identifying objects residing in locations—is the consistency of the set of the updates which have to be executed. The local description of global state changes also implies that by definition the next state differs from the previous state only at locations which appear in the update set. Simultaneous execution also provides a convenient way to abstract from sequentiality where it is irrelevant for the investigation. This synchronous parallelism in the ASM execution model is enhanced by the following notation to express the simultaneous execution of a rule R for each x satisfying a given condition φ (where typically x will have some free occurrences in R which are bound by the quantifier):

forall x **with** φ
  R

Similarly non-determinism as a convenient way to abstract from details of scheduling of rule executions can be expressed by rules of the form

choose x **with** φ
  R

where φ is a Boolean valued expression and R a rule. The meaning of such an ASM rule is to execute rule R with an arbitrary x chosen among those satisfying the selection property φ. If there exists no such x, nothing is done

Common notations like **where**, **let**, **if-then-else**, table-like **case** notation with pattern matching, etc. are used without further explanation since they are easily reducible to the above basic definitions. An occurrence of $r(x_1, \ldots, x_n)$ where a rule is expected stands for the corresponding rule R (which is supposed to be defined somewhere else, with $r(x_1, \ldots, x_n) = R$ appearing in the declaration part of the ASM where $r(x_1, \ldots, x_n)$ is used). When such a "rule call" $r(x_1, \ldots, x_n)$ is used, the parameters have to be instantiated by legal values (objects, functions, rules, whatever) so that the resulting rule has a well defined semantical meaning on the basis of the explanations given above. A precise semantical definition of such ASM submachine calls has been defined in [15].

For purposes of separation of concerns it is often convenient to impose for a given ASM additional constraints on its runs to circumscribe those one wants to consider as *legal*. Logically speaking this means to restrict the class of models satisfying the given specification. Such restrictions are particularly useful if the constraints express reasoning assumptions for a high-level machine which are easily shown to hold in a refined target machine. In general ASMs are reactive systems which iterate their computation step, but for the special case of terminating runs one can choose among various natural termination criteria to constrain runs, namely that no rule is applicable any more or that the machine yields an empty update set or that the state does not change any more.

5.2 Classification of Locations and Functions

In an ASM, a priori no restriction is imposed neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits however the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine.

The major distinction for a given ASM M is between its *static* functions—which never change during any run of M so that their values for given arguments do not depend on states of M—and *dynamic* ones which may change as a consequence of updates by M or by the environment (read: by some other—say an unknown—agent representing the context in which M computes), so that their values for given arguments may depend on states of M. By definition static functions can be thought of as given by the initial state, so that where appropriate, handling them can be clearly separated from the description of the system dynamics. Whether the meaning of these functions is determined by a mere signature ("interface") description, or by axiomatic constraints, or by an abstract specification, or by an explicit or recursive definition, or by a program module, depends on the degree of information hiding the specifier wants to realize. Static 0-ary functions represent constants, whereas with dynamic 0-ary functions one can model variables of programming (not to confuse with logical variables). Dynamic functions can be thought of as a generalization of array variables or hash tables.

The dynamic functions are further divided into four subclasses. *Controlled* functions (for M) are dynamic functions which are directly updatable by and only by the rules of M, i.e., functions f which appear in at least one rule of M as leftmost function (namely in an update $f(s) := t$ for some s, t) and are not updatable by the environment (or more generally by another agent in the case of a multi-agent machine). These functions are the ones which constitute the internally controlled part of the dynamic state of M.

Monitored functions, also called *in* functions, are dynamic functions which are read but not updated by M and directly updatable only by the environment (or more generally by other agents). They appear in updates of M, but not as leftmost function of an update. These monitored functions constitute the externally controlled part of the dynamic state of M. To describe combinations of internal and external control of functions, one can use *interaction* functions, also called *shared* functions, defined as dynamic functions which are directly updatable by rules of M and by the environment and can be read by both (so that typically a protocol is needed to guarantee consistency of updates). The concepts of monitored and shared functions allow one to separate in a specification computation from communication concerns. In fact the definition does not commit to any particular mechanism (e.g. message passing via channels) to describe the exchange of information between an agent and its environment (and similarly between arbitrary agents in the case of a multi-agent machine).

As with static functions the specification of monitored functions is open to any appropriate method. The only assumption made is that in a given state, the values of all monitored functions are determined.

Out functions are dynamic functions which are updated but not read by M and are monitored (read but not updated) by the environment or in general by other agents. Formally, such output functions do appear in some rules of M, but only as leftmost function of an assignment.

Functions are called *external* for M if for M they are either static or monitored.

An orthogonal, pragmatically important classification comes through the distinction of *basic* and of *derived* functions. Basic functions are functions which are taken for granted (declared as "given", typically those forming the basic signature); derived functions are functions which even if dynamic are not updatable neither by M nor by the environment but may be read by both and yield values which are defined by a fixed scheme in terms of other (static or dynamic) functions (and as a consequence may sometimes not be counted as part of the basic signature). Thus derived functions are sort of auxiliary functions coming with a specification or computation mechanism which is given separately from the main machine; they may be thought of as a global method with read-only variables.

The same classification principle is applied to (sets of) locations or updates.

5.3 Multi-Agent ASMs

A *multi-agent ASM* is defined as a set of agents which execute each its own basic ASM. This may happen in a synchronous or in an asynchronous manner. In a *synchronous ASM* the agents execute their basic ASM in parallel, synchronized using an implicit global system clock. Semantically a synchronous ASM is equivalent to the set of all its constituent single-agent ASMs, operating in the global states over the union of the signatures of each component, though each agent is equipped with its own set of states and rules. This concept allows one to define and analyze the interaction between components using precise interfaces over common locations.

A problem one has to solve for runs of asynchronously cooperating agents originates in the possible incomparability of their moves which may come with different data, clocks, moments and duration of execution, making it difficult to uniquely define a global state where moves are executed to locate changes of monitored functions in an ordering of moves. A coherence condition in the definition of asynchronous multi-agent ASM runs given in [21] postulates well-definedness for a relevant portion of state in which an agent is supposed to perform a step, thus providing a notion of 'local' stable view of 'the' state in which an agent makes a move. The underlying synchronization scheme is described using partial orders for moves of different agents which reflect causal dependencies, determining which move depends upon (and thus must come 'before') which other move. This synchronization scheme is as liberal as it can be, restricted only by the

consistency condition for the updates which is logically indispensable, and thus can be instantiated by any consistent synchronization mechanism.

Formally a run of an asynchronous ASM, also called *partial order run*, is defined as a partially ordered set $(M, <)$ of *moves* m (read: rule applications) of its agents satisfying the following conditions:

finite history: each move has only finitely many predecessors, i.e. for each $m \in M$ the set $\{m' \mid m' < m\}$ is finite,

sequentiality of agents: the set of moves $\{m \mid m \in M, a \text{ performs } m\}$ of every agent $a \in \mathit{Agent}$ is linearly ordered by $<$,

coherence: each finite initial segment (downward closed subset) X of $(M, <)$ has an associated state $\sigma(X)$—think of it as the result of all moves in X with m executed before m' if $m < m'$—which for every maximal element $m \in X$ is the result of applying move m in state $\sigma(X - \{m\})$.

The coherence condition immediately implies for every finite initial segment X of a run of an asynchronous ASM, all linearizations of X yield runs with the same final state. The definition provides no clue to construct partial order runs for an asynchronous ASM, but it makes the freedom explicit one has in implementing the described causal dependencies of certain local actions of otherwise independent agents. Notably the definition imposes no fairness condition on runs.