The ASM Method for System Design and Analysis. A Tutorial Introduction

Egon Börger

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

Abstract. We introduce into and survey the ASM method for high-level system design and analysis. We explain the three notions—*Abstract State Machine* [37], *ASM ground model* (system blueprint) [7] and *ASM refinement* [8]—that characterize the method, which integrates also current validation and verification techniques. We illustrate how the method allows the system engineer to rigorously capture requirements by ASM ground models and to stepwise refine these to code in a validatable and verifiable way.

1 Scope and Achievements of the ASM Method

An outstanding feature of the ASM method is that within a single *precise yet simple conceptual framework*, it naturally supports and uniformly integrates the following activities and techniques, as illustrated by Fig. 1 (taken from [24]):

– the major **software life cycle activities**, linking in a controllable way the two ends of the development of complex software systems:
  - **requirements capture** by constructing rigorous *ground models*, i.e. accurate concise high-level system blueprints (system contracts), formulated in domain-specific terms, using an application-oriented language which can be understood by all stakeholders [7],
  - **architectural and component design** bridging the gap between specification and code by *piecemeal, systematically documented detailing* of abstract models via stepwise refined models to code [8],
  - **validation** of models by their tool-supported *simulation*,
  - **verification** of model properties by tool-supported *proof techniques*,
  - **documentation** for *inspection*, *reuse* and *maintenance* by providing, through the intermediate models and their analysis, explicit descriptions of the *software structure* and of the major *design decisions*,

– the principal **modeling and analysis techniques**, on the basis of a systematic separation of different concerns (e.g. design from analysis, orthogonal design decisions, multiple levels of definitional or proof detail, etc.):
  - integrating dynamic (*operational*) and static (*declarative*) descriptions,
  - combining validation (simulation) and verification (proof) methods *at any desired level of detail.*

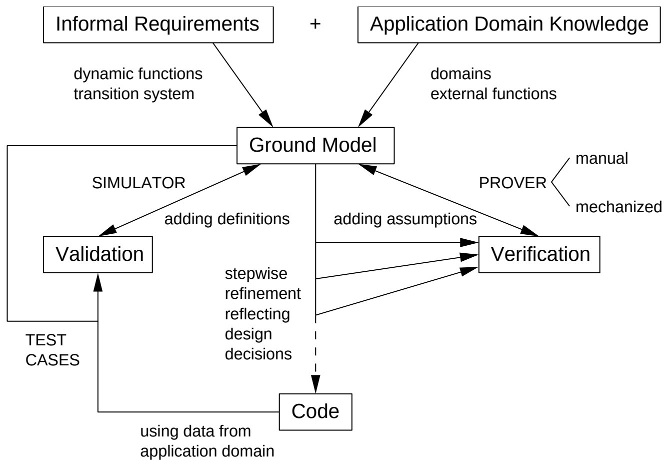


Fig. 1. Models and methods in the ASM-based development process

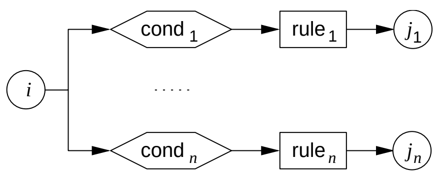
The integration potential of the ASM method is reflected by the great variety of its successful applications (for references see [24, 9]), for example:

- definition of industrial standards for Prolog (ISO), VHDL93 (IEEE), Java and JVM (Sun), SDL-2000 (ITU-T), C# (ECMA), BPEL for Web Services,
- design and reengineering of industrial control systems: software projects related to railway and mobile telephony network components (at Siemens), debugger and UPnP specification (at Microsoft), business systems interacting with intelligent devices (at SAP),
- modeling e-commerce and web services (at SAP),
- simulation and testing: a fire detection system in coal mines, the simulation of railway scenarios (at Siemens), the implementation of behavioral interface specifications on the .NET platform and conformence test of COM components (at Microsoft), compiler testing, test case generation,
- design and analysis of protocols for authentication, cryptography, cache-coherence, routing-layers for distributed mobile ad hoc networks, group-membership etc.,
- architectural design: verification (e.g. of pipelining schemes or of VHDL-based hardware design at Siemens), architecture/compiler co-exploration, combined validation and verification project,
- language design: definition, implementation and analysis of the semantics for real-life programmming languages, e.g. SystemC, Java/JVM—the book [51] contains the up to now most comprehensive non-proprietary real-life ASM case study, covering in every detail ground modeling, refinement, structuring, implementation, verification and validation of ASMs—, C#, domain-specific languages (Union Bank of Switzerland), etc.
- verification of compilation schemes and compiler back-ends.

The ASM method comes with a rigorous scientific foundation (see [24]). The *ASM ground model technique* adds the precision of mathematical blueprints to the loose character of human-centric UML descriptions. The *ASM refinement method* fills a widely-felt gap in UML-based techniques, namely by accurately linking the models at the successive stages of the system development cycle in an organic and effectively maintainable chain of coherent system views at different levels of abstraction. The resulting documentation maps the structure of the blueprint to compilable code, providing a road map for system use and maintenance. The practitioner needs no special training to use the ASM method since Abstract State Machines are a simple extension of Finite State Machines, obtained by replacing unstructured "internal" control states by states comprising arbitrarily complex data, and can be understood correctly as pseudo-code or Virtual Machines working over abstract data structures.

2 Turning FSMs into Abstract State Machines

In this section we explain ASMs as mathematical form of Virtual Machines that extend Finite State Machines and Codesign-FSMs by an enriched notion of state, which in support of modular design is accompanied by a classification of ASM locations defined below.[1]



```
**if** $ctl\_state = i$ **then**
  **if** $cond_1$ **then**
    $rule_1$
    $ctl\_state := j_1$
      ...
  **if** $cond_n$ **then**
    $rule_n$
    $ctl\_state := j_n$
```

Fig. 2. Viewing FSM instructions as control state ASM rules

An FSM is defined by a program of instructions of the form pictorially depicted in Fig. 2, where $i, j_1, \ldots, j_n$ are internal (control) states, $cond_\nu$ (for $1 \leq \nu \leq n$) represents the input condition $in = a_\nu$ (reading input a_ν) and $rule_\nu$ the output action $out := b_\nu$ (yielding output b_ν), which goes together with the *ctl_state* update to j_ν. Control state ASMs have the same form of programs, but the underlying notion of state is extended from three locations, namely:

– a single internal *ctl_state* assuming values in a not furthermore structured finite set

[1] The original definition in [37] was motivated by an epistemological concern related to the Church-Turing thesis. For historical details see [6]. The practice-oriented approach we follow here is taken from [10].

– two input and output locations *in*, *out* assuming values in a finite alphabet

to a *set of values of whatever types* residing in updatable memory units, so-called *locations*. Any desired level of abstraction can be achieved by permitting possibly parameterized locations to hold values of arbitrary complexity, whether atomic or structured: objects, sets, lists, tables, trees, graphs, whatever comes natural at the considered level of abstraction. As a consequence, the FSM updates of *ctl_state* and of its *out*put location are extended to ASM state changes resulting from updates of the value content of arbitrary many locations, namely via multiple assignments of the form $loc(x_1, \ldots, x_n) := val$.

This simple change of view of what a state is yields machines whose states can be arbitrary *multisorted structures*, i.e. domains of whatever objects coming with predicates (attributes) and functions defined on them, structures programmers nowadays are used to from object-oriented programming. In fact such a memory structure is easily obtained from the flat location view of abstract machine memory by grouping subsets of data into tables (arrays), via an association of a value to each table entry $(f, (a_1, \ldots, a_n))$. Here f plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of a table entry, $f(a_1, \ldots, a_n)$ denotes the value currently contained in the location $(f, (a_1, \ldots, a_n))$. Such a table represents an array variable f of dimension n, which can be viewed as the current interpretation of an n-ary "dynamic" function or predicate (boolean-valued function). This allows one to structure an ASM state as a set of tables and thus as a multisorted structure in the sense of mathematics.

In accordance with the extension of unstructured FSM control states to arbitrary ASM structures, the FSM-input *cond*itions are extended to arbitrary ASM-state expressions, which are called *guards* since they determine whether an instruction can be executed.[2] In addition, the usual non-deterministic interpretation, in case more than one FSM-instruction can be executed, is replaced by the parallel interpretation that in each ASM state, the machine executes simultaneously all the updates which are guarded by a condition that is true in this state. This *synchronous parallelism*, which yields a clear concept of *locally described global state change*, helps to abstract for high-level modeling from irrelevant sequentiality (read: an ordering of actions that are independent of each other in the intended design) and supports refinements to parallel or distributed implementations.

As a result of this extension of FSMs we obtain the definition of an ASM as a set of instructions of the following form, called ASM *rules* to stress the distinction between the parallel execution model for ASMs and the sequential single-instruction-execution model for traditional programs:

if *cond* **then** *Updates*

where *Updates* stands for a set of *function updates* $f(t_1, \ldots, f_n) := t$ built from expressions t_i, t and an n-ary function symbol f. The notion of run is the same

[2] For the special role of *in*/*out*put locations see below the classification of locations.

as for FSMs and for transition systems in general, taking into account the synchronous parallel interpretation.[3] Similarly to this extension of FSMs by basic ASMs, asynchronous ASMs extend globally asynchronous, locally synchronous Codesign-FSMs [42]. Only the notion of mono-agent sequential runs has to be extended to asynchronous (also called partially ordered) multi-agent runs. For a detailed definition in terms of ASMs we refer to [24, Ch.6.1].

Thus ASMs provide a rigorous mathematical semantics, which accurately supports the way application-domain experts use high-level process-oriented descriptions and software practitioners use "pseudo-code over abstract data". For the sake of completeness we list below notations for some other frequently used forms of rules, which enhance the expressivity of ASMs.

2.1 Classification of ASM Functions and Locations

In this section we describe how the ASM method supports the separation of concerns, information hiding, data abstraction, modularization and stepwise refinement by a systematic distinction between basic locations and derived ones (that are definable from basic ones), together with a read-write-permit classification of basic locations into static and dynamic ones and of the dynamic ones into monitored (only read), controlled (read and write), shared and output (only write) locations, as illustrated by Fig. 3.[4]

These distinctions reflect the different roles played in a given machine M by the auxiliary locations that are used in function updates to compute the arguments t_i and the new value t. The value of a *static* location never changes during any run of M because it does not depend on the states of M. The value of a *dynamic* location depends on the states of M since it may change as a consequence of updates either by M or by the environment. Static locations can be thought of as given by an initial system state, so that their definition can be treated separately from the description of the system dynamics. It depends on the degree of information-hiding the specifier wants to realize how the meaning of such locations is determined—by a mere signature ("interface") description or by axiomatic constraints or by an abstract specification, an explicit or recursive definition, a program module, etc.

Controlled locations for M are the ones which are directly updatable by and only by the rules of M, where they appear in at least one rule as the leftmost location in an update $f(s) := t$ for some s, t. These locations are the ones which constitute the internally controlled part of the dynamics of M, for example the location *ctl_state* in an FSM. Locations called *monitored* by M are those read but

[3] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *cond* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates if they are consistent. In the case of inconsistency, the run is considered as interrupted if no other stipulation is made, like calling an exception handling procedure or choosing a compatible update set.

[4] A set of locations or a function is called of a kind if all their locations are of that kind.

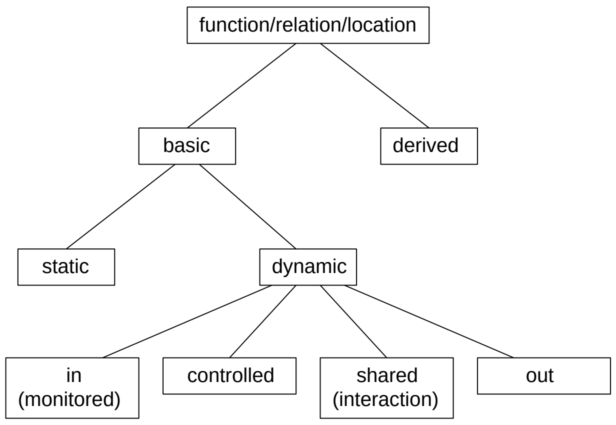


Fig. 3. Classification of ASM locations

not updated by M and updatable by other machines or the environment. They appear in updates of M, but not as a leftmost update location. An example is the input location *in* of an FSM. These monitored locations constitute the externally controlled part of the dynamic state of M. The concept of monitored locations allows one to separate in a specification the computation concerns from the 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 interacting agents. As with static locations the specification of monitored locations is open to any appropriate method, a feature that helps the system designer to control the amount of information which he wants to give to the programmer. The only (but crucial) assumption made is that in a given state the values of all monitored locations are determined.

Combinations of internal and external control are captured by *interaction* or *shared* locations that can be read and are directly updatable by more than one machine (so that typically a protocol is needed to guarantee consistency of updates). *Output* locations are updated but not read by M and are typically monitored by other machines or by the environment. An example is the location *out* of an FSM. Locations are called *external* for M if for M they are either static or monitored.

Distinguishing *basic* locations from *derived* locations whose values are defined by a fixed scheme in terms of other (static or dynamic) locations, pragmatically supports defining the latter by a specification or computation mechanism which is given separately from the main machine. Thus derived locations can be thought of as defining a global method with read-only variables.

An important type of monitored functions are dynamic selection functions f, which out of a collection X of objects satisfying a property φ select one element $f(X)$ in a way that may depend on the current state. They are frequently

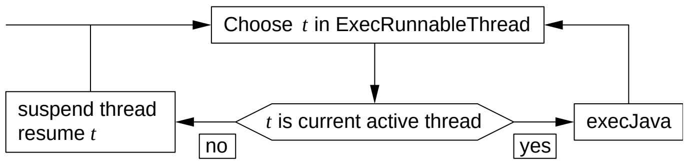


Fig. 4. Multiple thread Java machine

used to abstract from details of scheduling procedures. The following notation denotes $rule(f(X))$ when no specific name of the selection function f is needed:

```
choose $x$ with $\varphi$
  $rule(x)$
```

Also notational variations are frequently used, like **choose** $x \in X$ **in** $rule(x)$. Fig. 4 shows an example from the ASM model for thread handling in Java and C# [51, 49].

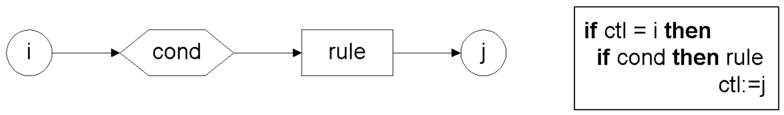
Similarly the following notation is used to make the synchronous parallelism of ASMs expressable in terms of arbitrary properties:

```
forall $x$ with $\varphi$
  $rule(x)$
```

standing for the simultaneous execution of $rule(x)$ for every element x satisfying φ.

2.2 Some Examples

Many industrial control systems, protocols, business processes and the like come with a concept of *status* or *mode* or *phase* that directs complex state transformations. Such a high-level system structure can be appropriately modeled by *control state ASMs*, introduced in [5] and closest to FSMs, i.e. ASMs all of whose rules are of the form in Fig. 5, written FSM(i, **if** *cond* **then** $rule, j$).



```
if ctl = i then
  if cond then rule
              ctl:=j
```

Fig. 5. FSM

A typical example is the top-level DEBUGGER model in Fig. 6, which was defined in [3] as part of a reverse-engineering case study to model a command-line

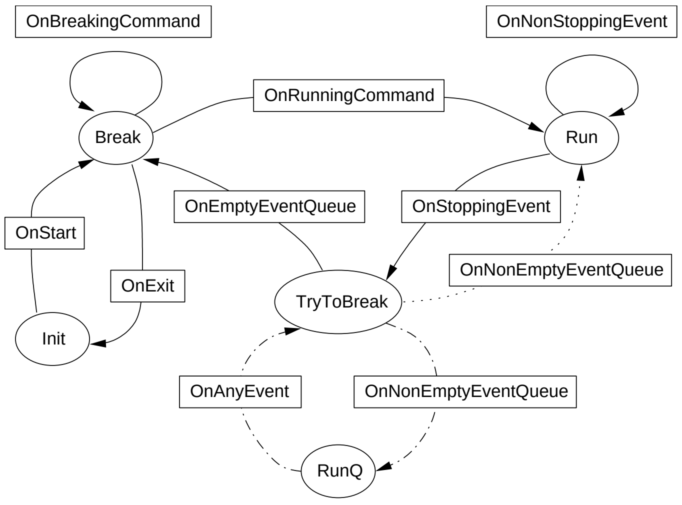


Fig. 6. DEBUGGER control state ASM

debugger of a stack-based run-time environment. During the reverse engineering process, this simple model led to the discovery of a flaw in the code, namely that the submachine executed during the dotted mode transition could lead to a deadlock and had to be replaced by a transition into a fifth mode *RunQ* (which was inserted into the implementation by an additional flag).

A business process example with only start/stop and busy mode is illustrated in Fig. 7, which is used in [1] to define the kernel of a web service mediator. The machine delivers for a current request a service answer that is to be compiled from the set of results of an iterative subrequest processing submachine, which in turn sends out further subrequests to – and collects the respective services from – other possibly independent subproviders.

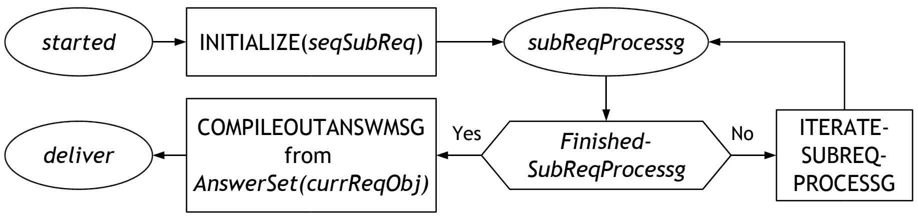


Fig. 7. Virtual Provider PROCESSing the current request

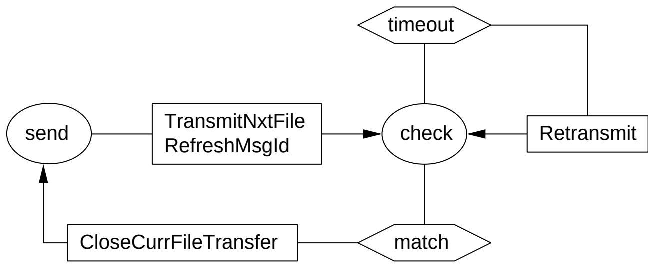


Fig. 8. Kermit protocol sender ASM

Fig. 8 defines the top-level control structure of a double-phase sender ASM, which appears in the Kermit protocol as AlternatingBitSender instance and as its refinement to a SlidingWindowSender [40]. For a generalization as a service interaction pattern see [4].

Fig. 9 from [23] defines the black-box view of neural nets characterized by two top-level phases: in the input phase the Neural Kernel is activated by the arrival of new input from the environment (transmitted by special input units to dedicated internal units), to perform on that input an internal computation which ends with emitting an output to the environment and switching back to the input mode.

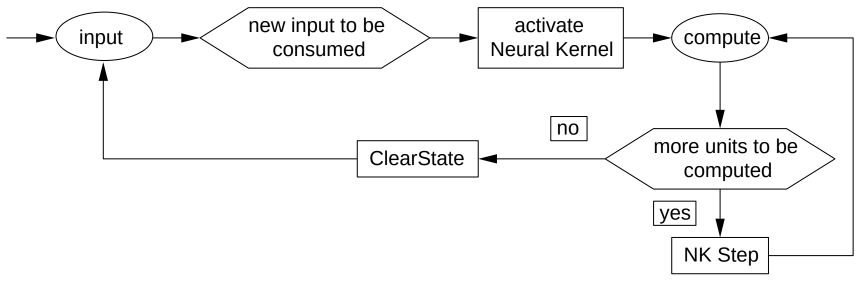


Fig. 9. Neural abstract machine model

2.3 ASM Submachines

The diagrams for control state ASMs enhance similar graphical UML notations by their rigorous semantics, which is formally defined in Fig. 2, 5, based upon the precise ASM semantics of the occurring abstract submachines that typically describe rather complex state transformations. In the examples above these submachines describe a Java interpreter *execJava* in Fig. 4; the Debugger actions

OnStart, etc. in Fig. 6; the subrequest processing iterator in Fig. 7; the different refinements of the Kermit macros in Fig. 8; the Neural Kernel Step submachine in Fig. 9 whose basic computing units (nodes of a directed data-flow graph) perform a finite sequence of atomic actions propagating their results through the graph until the output units are reached.

Where convenient one can also abstract away the FSM-typical control-state details of an intended sequentiality and encapsulate the execution of a machine M immediately followed by the execution of N into a black-box view M **seq** N, which is supported also by the well-known traditional graphical representations of FSMs that omit labels for intermediate internal states. Iterating such a **seq** operator leads to so-called turbo ASMs that support the standard iteration constructs of programming. In the same way one can define a general ASM submachine concept that fits the synchronous parallel view of ASMs and supports the two abstraction levels defined by the black-box and the white-box view of submachines (see [21]). It also supports the traditional understanding of recursive machine calls (see [12]).

We illustrate ASM submachines by two examples. The first one is the submachine INITIALIZE(*class*) used in the ASM model for a Java interpreter [51], providing a succinct formulation for the intricate interaction of the initialization of classes with other language concepts. In Java the initialization of a class c is done implicitly at the first use of c, respecting the class hierarchy (the superclass of c has to be initialized before c). Thus INITIALIZE(*class*) stores its call parameter *class*, say into a local variable *currInitClass*, and then iterates the creation of class initialization frames until a class is reached which is *Initialized*.[5]

```
INITIALIZE($class$) =
  $currInitClass := class$ seq
    while not $Initialized(currInitClass)$
      CREATEINITFRAME($currInitClass$)
      if not $Initialized(superClass(currInitClass))$ then
        $currInitClass := superClass(currInitClass))$
```

The INITIALIZE submachine offers the possibility that the designer works with a black-box view—of an atomic operation that pushes all initialization methods in the right order onto the frame stack, followed by calling the Java interpreter to execute them (in the inverse order)—whereas the programmer and the verifier work with the refined white-box view, which provides the necessary details to implement the machine and to analyze its global properties of interest (see [22]). A refinement of INITIALIZE for a C# interpreter has been defined in [17] and has been used in [32] to investigate problems related to class initialization in C#.

We illustrate the support of recursive submachines by an ASM describing the well-known procedure to quicksort lists L: FIRST partition the *tail* of the list

[5] The termination happens at the latest at the top of the finite class hierarchy. The submachine CREATEINITFRAME(c) sets $classState(c)$ to $InProgress$ whereby $Initialized(currInitClass)$ becomes true.

into the two sublists $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ of elements $< head(L)$ respectively $\geq head(L)$ and quicksort these two sublists separately (independently of each other), THEN *concatenate* the results placing $head(L)$ between them.[6]

```
$\text{Quicksort}(L) =$
  $\textbf{if} \mid L \mid \leq 1$ then result := $L$ else
    let
      $x = \text{Quicksort}(tail(L)_{<head(L)})$
      $y = \text{Quicksort}(tail(L)_{\geq head(L)})$
    in result := $concatenate(x, head(L), y)$
```

Computing $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ appears in this machine as an external subcomputation. We illustrate in Sect. 5 how to internalize such a subcomputation by a refinement step.

3 ASM Ground Models (System Blueprints)

The role of a system blueprint (ground model) is to capture changing system requirements ("what to build") in a *consistent and unambiguous*, *simple and concise*, *abstract and complete* way, so that the resulting documentation "grounds the design in reality" by its being *understandable and checkable* (for correctness and completeness) by both domain experts and system designers. Using ASMs one can cope with ever-changing requirements by building ground models for change which share the above eight attributes, as we will shortly describe here, refering for further explanations to [7].

Understandability implies that domain expert and system designer share the language in which the ground model is formulated, as part of the contract that binds the two parties. In this respect it is crucial that ASMs allow one to calibrate the degree of precision of a ground model to the conceptual frame of the given problem domain, supporting the concentration on domain issues instead of issues of notation.

Checkability means that both reasoning and experimentation can be applied to a blueprint to establish that it 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. Since ASM ground models are formulated in application-domain terms, they are inspectable for correctness and completeness by the application-domain expert; on the other side, due to their mathematical nature, they also support the designer in mathematically checking the internal model consistency and the consistency of different system views. In addition, exploiting the concept of ASM run, one can perform experiments with ASM ground models simulating them for running relevant scenarios (use cases), supporting systematic attempts to "falsify" the model against the to-be-encoded piece of reality. As technical side-effect one can define – prior to coding – a precise system-acceptance test plan, thus turning

[6] See [12] for a formal definition of the **let** $x = R(a), y = S(b)$ **in** M construct.

the ground model into a test model that is to be matched by the tester against executions of the final code.

Understandability and checkability of ASM ground models already help to avoid that a software project fails simply because it does not build the right system, due to a misunderstanding of the requirements. We now shortly characterize the remaining above mentioned six intrinsic properties an ASM ground model has to satisfy, namely to be:

– *precise* (unambiguous and consistent) at the appropriate level of detailing yet *flexible*, to satisfy the required accuracy avoiding unnecessary precision;
– *simple and concise* to be understandable by both domain experts (for inspection) and system designers (for analysis). ASMs allow one to explicitly formulate those abstractions that “directly” reflect the structure of the real-world problem, avoiding any extraneous encoding;
– *abstract (minimal) yet complete*. Completeness means that all and only semantically relevant features are to be made present: parameters concerning the interaction with the environment, the basic architectural system structure, the domain knowledge representation, etc., alltogether making the ASM “closed” modulo some “holes”. However, the holes are explicitly delineated, including statements of the assumptions made for them at the abstract level (to be realized through the detailed specification via later refinements). 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.

It is this combination of blueprint properties that made ASM ground models so successful as means to formulate high-level models for industrial control systems, patent documents, standards. See the formulation of the forthcoming standard for the Business Process Execution Language for Web Services [52], for the ITU-T standard for SDL-2000 [35], the ECMA standard for C# [17], the de facto standard for Java and its implementation on the JVM [51], the IEEE-VHDL93 standard [18], the ISO-Prolog standard [14]. Or see the development of railway [13, 19] and mobile telephony network components [25] at Siemens. These examples show also that ASM ground models are fit for *reuse*. When the requirements change, these changes can often be directly reflected by blueprint adaptations, typically additions to or refinements of the ground model abstractions.

4 ASM Refinements (Reflecting Design Decisions)

We describe in this section the practice-oriented *ASM refinement notion* [8], which provides a framework to systematically separate, structure and document orthogonal design decisions and thus to effectively relate different system views and aspects. The method supports cost-effective system maintenance and management of system changes as well as piecemeal system validation and verification techniques. Putting together the single refinement steps, typically into a chain or tree of successively more detailed models, allows the designer to rigorously link the system architect's view (at the abstraction level of a blueprint) to the programmer's view (at the level of detail of compilable code), crossing levels of abstraction in a way that supports design-for-change.

Refinement is a general methodological principle which is present wherever a complex system or problem is described piecemeal, decomposing it into constituent parts which are detailed in steps to become manageable. Refinement goes together with the inverse process of abstraction. The principle of the ASM refinement method is illustrated by Fig. 10: to refine an ASM M to an ASM M^*, the designer has the freedom to define the following items:

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

Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states. More precisely: fix any notions $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

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

Often the M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$. The states $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) in Fig. 10, for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest). The scheme shows that an ASM refinement allows one to combine in a natural way a change of the signature (through the

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

With an equivalence notion $\equiv$ between data in locations of interest in corresponding states.

Fig. 10. The ASM refinement scheme

definition of states and of their correspondence, of corresponding locations and of the equivalence of data) with a change of the control (defining the "flow of operations" appearing in the corresponding computation segments).

It is important for the practicability of ASM refinements that the size of m and n in (m, n)-refinements is allowed to dynamically depend on the state. Practical experience also shows that (m, n)-refinements with $n > 1$ and including $(m, 0), (0, n)$-steps support the feasibility of decomposing complex (global) actions into simpler (locally describable) ones whose behavior can be verified in practice. Procedural $(1, n)$-refinements with $n > 1$ have their typical use in compiler verification when replacing a source code instruction by a chunk of target code; for numerous examples see [16, 15, 53, 39, 36, 51]. A convenient way to hide multiple steps of a procedural refinement is to use ASM submachines as discussed above, which allow one to "view" n submachine steps as one step of an overall (here the more abstract) computation.

The ASM literature surveyed in [6] is full of examples of the above definition, which generalizes numerous more specialized and less practical refinements notions in the literature [43, 44]. The ASM refinement method integrates declarative and operational techniques and widely used modularization concepts into the design and analysis of ASM models. In particular it supports modularizing ASM refinement correctness proofs aimed at mechanizable proof support, see [43].

5 ASM Analysis Techniques (Validation and Verification)

Based upon the notion of ASM *run*, various tools have been built to mechanically execute ASM models for their experimental validation by simulation and testing, notably: *ASM Workbench* [26], *AsmGofer* [46], ASM2C++ compiler [47], *XASM* [2], *AsmL* [31] and *CoreASM* [30]. Based upon the mathematical character of ASMs, also any standard mathematical verification techniques can be applied to prove or disprove ASM model properties, implying precision at the desired level of rigour: from proof sketches over traditional [20, 51] or formalized mathematical proofs [50] to tool supported proof checking or interactive or automatic theorem proving, e.g. by KIV [45], PVS [28, 33], model checkers [27, 34]. In a comprehensive development and analysis environment for real-life ASMs, various combinations of such verification and validation methods can be supported and can be used for the analysis of compilers [29, 41] and hardware [48, 38] and in the context of the program verifier challenge [11].

6 Combined Refinement and Verification Example

In this section we illustrate for the mathematically inclined reader how to combine the stepwise refinement technique with piecemeal proving of properties of interest. We use as simple but characteristic examples a refinement of the above QUICKSORT machine and an ASM for the well-known leader election protocol together with its extension by a shortest path computation.

The goal of the leader election protocol is to achieve the election of a leader among finitely many homogeneous agents in a connected network, using only communication between neighbor nodes. The *leader* is defined as $max(Agent)$ with respect to a linear order $<$ among agents. The algorithmic idea, underlying the ASM defined in Fig. 11 together with the macros below, is as follows: every agent proposes to his *neighb*ors his current leader *cand*idate, checks the leader *proposals* received from his *neighb*ors and upon detecting a proposal which improves his leader candidate, he improves his candidate for his next proposal. The protocol correctness to be proved reads as follows: if initially every agent is without *proposals* from his neighbors and will *proposeToNeighbors* him*self* as *cand*idate, then eventually every agent will *checkProposals* with empty set *proposals* and $cand = max(Agent)$.

```
LEADERELECTIONMACROS =
   $propose$ = forall $n \in neighb$ insert $cand$ to $proposals(n)$
   $proposals$ improve = $max(proposals) > cand$
   improve by $proposals$ = $cand := max(proposals)$
   $EmptyProposals = (proposals := empty)$
   there are proposals = $(proposals \neq empty)$
```

Assuming that every enabled agent will eventually make a move, the protocol correctness can be proved by an induction on runs and on $\sum\{leader - cand(n) \mid n \in Agent\}$, which measures the distances of candidates from the leader.

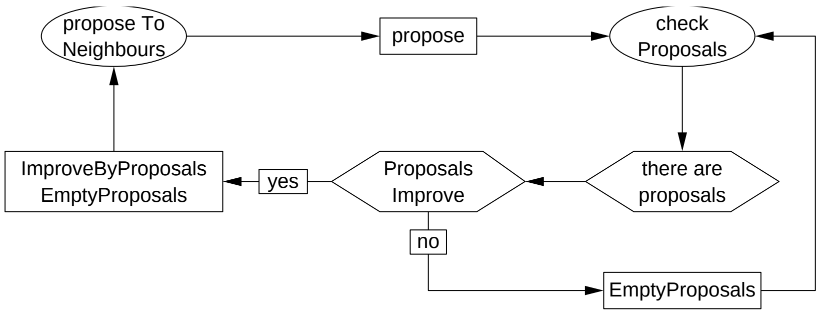


Fig. 11. Basic ASM of LeaderElection agents

Assume we now want to compute for each agent also a shortest path to the leader. One has to provide for every agent (except for the leader), in addition to the leader candidate, also a neighbor which is currently known to be closest to the leader, together with the minimal distance to the leader via that neighbor. This is an example of a pure data refinement and consists in enriching *cand* and *proposals* by a neighbor with minimal distance to the leader, recorded in new dynamic functions $nearNeighb$: $Agent$ and $distance$: $Distance$ (e.g. $Distance = \mathbb{N} \cup \{\infty\}$), so that $proposals \subseteq Agent \times Agent \times Distance$ (triples of leader *cand*, *nearNeighbor* and *distance* to the leader). Initially we assume $nearNeighbor =$ **self** and $distance = \infty$ except for the *leader* where $distance = 0$.

Thus each agent of the refined async MinPathToLeader ASM executes the properly initialized basic ASM defined in Fig. 11 with the refined macros below. Priority is given to determine the largest among the proposed neighbors (where *Max* over triples takes the *max* over the proposed neighbor agents), among the *proposalsFor* the current *cand* the one with *minimal distance* is chosen.

```
MinPathToLeaderMacros =
  $propose =$ forall $n \in neighb$
      insert $(cand, nearNeighb, distance)$ to $proposals(n)$
  $proposals$ improve $=$ let $m = Max(proposals)$ in
    $m > cand$ or
      $(m = cand$ and $minDistance(proposalsFor\ m) + 1 < distance)$
  improve by $proposals =$
    $cand := Max(proposals)$
    update PathInfo to $Max(proposals)$
  update PathInfo to $m =$choose $(n, d)$ with
    $(m, n, d) \in proposals$ and $d = minDistance(proposalsFor\ m)$
      $nearNeighb := n$
      $distance := d + 1$
```

The leader election correctness property can now be sharpened by the shortest path correctness property, namely that eventually for every agent, *distance* is the minimal distance of a path from agent to leader, and *nearNeighbor* is a neighbor on a minimal path to the leader (except for the leader where *nearNeighbor* = *leader*). The proof extends the above induction by a side induction on the minimal distances in *proposalsFor Max*(*proposals*.

As second example we illustrate how by a refinement step for QUICKSORT one can internalize the computation of $tail(L)_{<head(L)}, tail(L)_{\geq head(L)}$ into a partitioning submachine PARTITION(l, h, p). This machine works on the representation of lists as functions $L\colon [r, s] \to VAL$ from intervals of natural numbers to a set of values. When $r < s$, *Partition* is started with the search boundaries $l = r, h = s$ and the list head $pivot = L(r)$. It terminates when reaching $l = h$ with $L(l) = pivot$, all L-elements smaller than the pivot to the left of l, and all the others at l or to the right of l. Until reaching $l = h$, the partitioning procedure alternates between searching from above for list elements $L(h) \leq pivot$ and searching from below for list elements $L(l) \geq pivot$. When such an element is encountered and it is different from the element at the other current search boundary—one of them is the pivot—, then the boundary elements $L(l), L(h)$ are swapped and the search switches to the other boundary. When $L(h) \leq pivot \leq L(l) \leq L(h)$ before $l = h$ is encountered (namely when pivot has multiple occurrences in the list), h can be decreased by one.

```
PARTITION$(l, h, pivot)$ =
  **if** $L(h) > pivot$ **then** $h := h - 1$
  **elseif** $L(l) < pivot$ **then** $l := l + 1$
  **elseif** $L(l) > L(h)$ **then**
    $L(l) := L(h)$
    $L(h) := L(l)$
  **elseif** $l < h$ **then** $h := h - 1$
```

7 Conclusion

The ASM method offers no fool-proof button-pushing, completely mechanical design and verification procedure, but it directly supports professional knowledge and skill in "building models for change", stepwise detailing them to compilable code and maintaining models and code in a cost-effective and reliable way. This is the best one can hope for, given the intrinsically creative character of defining the essence of a complex computer-based system.[7]

[7] Final version to appear in B. Gramlich (Ed.): Frontiers of Combining Systems. Springer LNAI 3717 (2005), 264-283.

References

1. M. Altenhofen, E. Börger, and J. Lemcke. A high-level specification for mediators. In *1st International Workshop on Web Service Choreography and Orchestration for Business Process Management*, 2005.
2. M. Anlauff and P. Kutter. Xasm Open Source. Web pages at http://www.xasm.org/, 2001.
3. M. Barnett, E. Börger, Y. Gurevich, W. Schulte, and M. Veanes. Using Abstract State Machines at Microsoft: A case study. 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 367–380. Springer-Verlag, 2000.
4. A. Barros and E. Börger. A compositional framework for service interaction patterns and communication flows. In *Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, Springer LNCS, 2005.
5. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.
6. E. Börger. The origins and the development of the ASM method for high-level system design and analysis. *J. Universal Computer Science*, 8(1):2–74, 2002.
7. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
8. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
9. E. Börger. Modeling with Abstract State Machines: A support for accurate system design and analysis. In B. Rumpe and W. Hesse, editors, *Modellierung 2004*, volume P-45 of *GI-Edition Lecture Notes in Informatics*, pages 235–239. Springer-Verlag, 2004.
10. E. Börger. From finite state machines to virtual machines (Illustrating design patterns and event-B models). In E. Cohors-Fresenborg and I. Schwank, editors, *Präzisionswerkzeug Logik–Gedenkschrift zu Ehren von Dieter Rödding*. Forschungsinstitut für Mathematikdidaktik Osnabrück, 2005. ISBN 3-925386-56-4.
11. E. Börger. Linking content definition and analysis to what the compiler can verify. In *Proc.IFIP WG Conference on Verified Software: Tools, Techniques, and Experiments*, Lecture Notes in Computer Science, Zurich (Switzerland), October 2005. Springer.
12. E. Börger and T. Bolognesi. Remarks on turbo ASMs for computing functional equations and recursion schemes. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003 – Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 218–228. Springer-Verlag, 2003.
13. E. Börger, H. Busch, J. Cuellar, P. Päppinghaus, E. Tiden, and I. Wildgruber. Konzept einer hierarchischen Erweiterung von EURIS. Siemens ZFE T SE 1 Internal Report BBCPTW91-1 (pp. 1–43), Summer 1996.
14. 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.
15. E. Börger and G. Del Castillo. A formal method for provably correct composition of a real-life processor out of basic components (The APE100 Reverse Engineering

Study). In B. Werner, editor, *Proc. 1st IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS'95)*, pages 145–148, November 1995.

16. E. Börger and I. Durdanović. Correctness of compiling Occam to Transputer code. *Computer Journal*, 39(1):52–92, 1996.
17. 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), 2005.
18. E. Börger, U. Glässer, and W. Müller. The semantics of behavioral VHDL'93 descriptions. In *EURO-DAC'94. European Design Automation Conference with EURO-VHDL'94*, pages 500–505, Los Alamitos, California, 1994. IEEE Computer Society Press.
19. 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.
20. 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.
21. 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 *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
22. E. Börger and W. Schulte. Initialization problems for Java. *Software – Concepts and Tools*, 19(4):175–178, 2000.
23. E. Börger and D. Sona. A neural abstract machine. *J. Universal Computer Science*, 7(11):1007–1024, 2001.
24. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
25. G. D. Castillo and P. Päppinghaus. Designing software for internet telephony: experiences in an industrial development process. In A. Blass, E. Börger, and Y. Gurevich, editors, *Theory and Applications of Abstract State Machines*, Schloss Dagstuhl, Int. Conf. and Research Center for Computer Science, 2002.
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. .
27. G. Del Castillo and K. Winter. Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, *Proc. 6th Int. Conf. TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 331–346. Springer-Verlag, 2000.
28. A. Dold. A formal representation of Abstract State Machines using PVS. Verifix Technical Report Ulm/6.2, Universität Ulm, Germany, July 1998.
29. A. Dold, T. Gaul, V. Vialard, and W. Zimmermann. ASM-based mechanized verification of compiler back-ends. In U. Glässer and P. Schmitt, editors, *Proc. 5th Int. Workshop on Abstract State Machines*, pages 50–67. Magdeburg University, 1998.
30. R. Farahbod, V. Gervasi, and U. Glässer. CoreASM: An extensible ASM execution engine. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05.* Université de Paris 12, 2005.
31. Foundations of Software Engineering Group, Microsoft Research. AsmL. Web pages at http://research.microsoft.com/foundations/AsmL/, 2001.

32. N. G. Fruja. Specification and implementation problems for C#. In B. Thalheim and W. Zimmermann, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2004.
33. A. Gargantini and E. Riccobene. Encoding Abstract State Machines in PVS. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 303–322. Springer-Verlag, 2000.
34. A. Gawanmeh, S. Tahar, and K. Winter. Interfacing ASMs with the MDG tool. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, pages 278–292. Springer-Verlag, 2003.
35. U. Glässer, R. Gotzhein, and A. Prinz. Formal semantics of SDL-2000: Status and perspectives. *Computer Networks*, 42(3):343–358, June 2003.
36. G. Goos and W. Zimmermann. Verifying compilers and ASMs. In Y. Gurevich, P. Kutter, M. Odersky, and L. Thiele, editors, *Abstract State Machines: Theory and Applications*, volume 1912 of *Lecture Notes in Computer Science*, pages 177–202. Springer-Verlag, 2000.
37. Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
38. A. Habibi. *Framework for System Level Verification: The SystemC Case*. PhD thesis, Concordia University, Montreal, July 2005.
39. A. Heberle. *Korrekte Transformationsphase – der Kern korrekter Übersetzer*. PhD thesis, Universität Karlsruhe, Germany, 2000.
40. J. Huggins. Kermit: Specification and verification. In E. Börger, editor, *Specification and Validation Methods*, pages 247–293. Oxford University Press, 1995.
41. A. Kalinov, A. Kossatchev, A. Petrenko, M. Posypkin, and V. Shishkov. Using ASM specifications for compiler testing. In E. Börger, A. Gargantini, and E. Riccobene, editors, *Abstract State Machines 2003–Advances in Theory and Applications*, volume 2589 of *Lecture Notes in Computer Science*, page 415. Springer-Verlag, 2003.
42. L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer-Verlag, 2000.
43. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
44. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 2004.
45. G. Schellhorn and W. Ahrendt. Reasoning about Abstract State Machines: The WAM case study. *J. Universal Computer Science*, 3(4):377–413, 1997.
46. J. Schmid. Executing ASM specifications with AsmGofer. Web pages at http://www.tydo.de/AsmGofer.
47. J. Schmid. Compiling Abstract State Machines to C++. *J. Universal Computer Science*, 7(11):1069–1088, 2001.
48. J. Schmid. *Refinement and Implementation Techniques for Abstract State Machines*. PhD thesis, University of Ulm, Germany, 2002.
49. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer-Verlag, 2004.
50. R. F. Stärk and S. Nanchen. A logic for Abstract State Machines. *J. Universal Computer Science*, 7(11):981–1006, 2001.

51. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation.* Springer-Verlag, 2001. .
52. M. Vajihollahi. High level specification and validation of the Business Process Execution Language for web services. Master's thesis, School of Computing Science at Simon Fraser University, April 2004.
53. 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.